Design and Analysis of Multidimensional Data Structures

Tesi doctoral presentada al
Departament de Llenguatges i Sistemes Informàtics
de la Universitat Politècnica de Catalunya

per optar al grau de
Doctora en Informàtica

per
Amalia Duch Brown

sota la direcció del doctor
Conrado Martínez Parra

Barcelona, 25 d'octubre de 2004

Aquesta tesi fou llegida el dia 9 de desembre de 2004, davant el tribunal de tesi format per:

- Dr. Luc Devroye (President)
- Dr. Salvador Roura (Secretari)
- Dr. Ricardo Baeza-Yates
- Dr. Pere Brunet
- Dr. Ralph Neininger

Let no one say that I have said nothing original, at least the arrangement of the subject is new.

—Blaise Pascal: Pensées.

RESUM

Aquesta tesi està dedicada al disseny i a l'anàlisi d'estructures de dades multidimensionals, és a dir, estructures de dades que serveixen per emmagatzemar registres K-dimensionals que solen representar-se com a punts en l'espai $[0,1]^K$. Aquestes estructures tenen aplicacions en diverses àrees de la informàtica com poden ser els sistemes d'informació geogràfica, la robòtica, el processament d'imatges, la world wide web, el data mining, entre d'altres. Les estructures de dades multidimensionals també es poden utilitzar com a indexos d'estructures de dades que emmagatzemen, possiblement en memòria externa, dades més complexes que els punts.

Les estructures de dades multidimensionals han d'oferir la possibilitat de realitzar operacions d'inserció i esborrat de claus dinàmicament, a més de permetre realitzar cerques anomenades *associatives*. Exemples d'aquest tipus de cerques són les cerques per *rangs ortogonals* (quins punts cauen dintre d'un hiper-rectangle donat?) i les cerques del *veí més proper* (quin és el punt més proper a un punt donat?).

Podem dividir les contribucions d'aquesta tesi en dues parts:

1. La primera part està relacionada amb el disseny d'estructures de dades per a punts multidimensionals. Inclou el disseny d'arbres binaris K-dimensionals al·leatoritzats (*Randomized K-d trees*), el d'arbres quaternaris al·leatoritzats (*Randomized quad trees*) i el d'arbres multidimensionals amb punters de referència (*Fingered multidimensional trees*).

2. La segona part analitza el comportament de les estructures de dades multidimensionals. En particular, s'analitza el cost mitjà de les cerques parcials en arbres K-dimensionals relaxats, i el de les cerques per rang en diverses estructures de dades multidimensionals.

Respecte al disseny d'estructures de dades multidimensionals, proposem algorismes al·leatoritzats d'inserció i esborrat de registres per als arbres K-dimensionals i per als arbres quaternaris. Aquests algorismes produeixen arbres aleatoris, independenment de l'ordre d'inserció dels registres i després de qualsevol seqüència d'insercions i esborrats. De fet, el comportament esperat de les estructures produïdes mitjançant els algorismes al·leatoritzats és independent de la distribució de les dades d'entrada, tot i conservant la simplicitat i la flexibilitat dels arbres K-dimensionals i quaternaris estàndard. Introduïm també els arbres multidimensionals amb punters de referència. Això permet que les estructures de dades multidimensionals puguin aprofitar

l'anomenada localitat de referència en cerques associativas altament correlacionades.

I respecte de l'anàlisi d'estructures de dades multidimensionals, primer analitzem el cost esperat de las cerques parcials en els arbres K-dimensionals relaxats. Seguidament utilitzem aquest resultat com a base per a l'anàlisi de les cerques per rangs ortogonals, juntament amb arguments combinatoris i geomètrics. D'aquesta manera obtenim un estimat asimptòtic precís del cost de les cerques per rangs ortogonals en els arbres K-dimensionals aleatoris.

Finalment, mostrem que les tècniques utilitzades es poden extendre fàcilment a d'altres estructures de dades i per tant proporcionem una anàlisi del cost mitjà de cerques per rang en estructures de dades com són els arbres K-dimensionals estàndard, els arbres quaternaris, els *tries* quaternaris i els *tries* K-dimensionals.

RESUMEN

Esta tesis está dedicada al diseño y al análisis de estructuras de datos multidimensionales; es decir, estructuras de datos específicas para almacenar registros K-dimensionales que suelen representarse como puntos en el espacio $[0,1]^K$. Estas estructuras de datos tienen aplicaciones en diversas áreas de la informática como son: los sistemas de información geográfica, la robótica, el procesamiento de imágenes, la world wide web o data mining, entre otras. Las estructuras de datos multidimensionales suelen utilizarse también como índices de estructuras que almacenan, posiblemente en memoria externa, datos más complejos que los puntos. Las estructuras de datos multidimensionales deben ofrecer la posibilidad de realizar operaciones de inserción y borrado de llaves de manera dinámica, pero además deben permitir realizar *búsquedas asociativas* en los registros almacenados. Ejemplos de búsquedas asociativas son las *búsquedas por rangos ortogonales* (¿qué puntos de la estructura de datos están dentro de un hiper-rectángulo dado?) y las *búsquedas del vecino más cercano* (¿cuál es el punto de la estructura de datos más cercano a un punto dado?).

Las contribuciones de esta tesis se dividen en dos partes:

1. La primera parte está dedicada al diseño de estructuras de datos para puntos multidimensionales e incluye el diseño de los árboles binarios K-dimensionales aleatorizados (*Randomized K-d trees*), el de los árboles cuaternarios aleatorizados (*Randomized quad trees*), y el de los árboles multidimensionales con punteros de referencia (*Fingered multidimensional trees*).

2. La segunda parte contiene contribuciones al análisis del comportamiento de las estructuras de datos multidimensionales. En particular, damos el análisis del costo promedio de las búsquedas parciales en los árboles K-dimensionales relajados y el de las búsquedas por rango en varias estructuras de datos multidimensionales.

Con respecto al diseño de estructuras de datos multidimensionales, proponemos algoritmos aleatorios de inserción y borrado de registros para los árboles K-dimensionales y los árboles cuaternarios que producen árboles aleatorios independientemente del orden de inserción de los registros y después de cualquier secuencia de inserciones y borrados intercalados. De hecho, con la aleatorización, garantizamos un buen rendimiento esperado de las estructuras de datos resultantes que es independiente de la distribución de los

datos de entrada, conservando la flexibilidad y la simplicidad de los árboles K-dimensionales y de los árboles cuaternarios estándar. En esta parte proponemos también los árboles multidimensionales con punteros de referencia, una técnica que permite que las estructuras de datos multidimensionales exploten la localidad de referencia en búsquedas asociativas que se presentan altamente correlacionadas.

Con respecto al análisis de estructuras de datos multidimensionales, comenzamos dando un análisis preciso del costo esperado de las búsquedas parciales en los árboles K-dimensionales relajados. A continuación, utilizamos este resultado como base para el análisis de las búsquedas por rangos ortogonales, combinándolo con argumentos geométricos y combinatorios. Como resultado obtenemos un estimado asintótico preciso del costo de las búsquedas por rango en los árboles K-dimensionales relajados.

Finalmente, mostramos que las técnicas utilizadas pueden extenderse fácilmente a otras estructuras de datos y por tanto proporcionamos un análisis del costo promedio de búsquedas por rango en estructuras de datos como los árboles K-dimensionales éstandar, los árboles cuaternarios, los *tries* K-dimensionales y los *tries* cuaternarios.

ABSTRACT

This thesis is about the design and analysis of point multidimensional data structures: data structures that store K-dimensional keys which we may abstract as points in $[0,1]^K$. These data structures are present in many applications of geographical information systems, image processing, robotics, world wide web, or data mining among others. They are also frequently used as indexes of more complex data structures, possibly stored in external memory. Point multidimensional data structures must have capabilities such as insertion, deletion and (exact) search of items, but in addition they must support the so called *associative queries*. Examples of these queries are orthogonal range queries (which are the items that fall inside a given hyper-rectangle?) and nearest neighbor queries (which is the closest item to some given point q?).

The contributions of this thesis are two-fold:

1. Contributions to the design of point multidimensional data structures which includes: the design of randomized K-d trees, the design of randomized quad trees and the design of fingered multidimensional search trees;

2. Contributions to the analysis of the performance of point multidimensional data structures: the average-case analysis of partial match queries in relaxed K-d trees and the average-case analysis of orthogonal range queries in various multidimensional data structures.

Concerning the design of randomized point multidimensional data structures, we propose randomized insertion and deletion algorithms for K-d trees and quad trees that produce random K-d trees and quad trees independently of the order in which items are inserted into them and after any sequence of interleaved insertions and deletions. The use of randomization provides expected performance guarantees, irrespective of any assumption on the data distribution, while retaining the simplicity and flexibility of standard K-d trees and quad trees.

Also related to the design of point multidimensional data structures is the proposal of fingered multidimensional search trees, a new technique that enhances point multidimensional data structures to exploit locality of reference in associative queries.

With regards to performance analysis, we start by giving a precise analysis of the cost of partial matches in randomized K-d trees. We use these results

as a building block in our analysis of orthogonal range queries, together with combinatorial and geometric arguments and we provide a tight asymptotic estimate of the cost of orthogonal range search in randomized K-d trees. We finally show that the techniques used apply easily to other data structures, so we can provide a precise analysis of the average cost of orthogonal range search in other data structures such as standard K-d trees, quad trees, quad tries, and K-d tries.

ACKNOWLEDGMENTS

I once heard –possibly in a quotation by Thomas Fuller– that *"courage may triumph over uncertain questions, while only patience over desperate ones"*. The completion of this dissertation involved a long list of desperate situations that I prefer not to mention here. However, I wish to express my gratitude to the many teachers, friends and family who helped and encouraged me through their guidance, example, support and, mostly, patience.

I want to thank, specially, Conrado Martínez whose advice, unlimited patience, guidance and constant encouragement (often beyond academic matters) made this thesis possible. Salvador Roura, whose work is the foundation of mine. Josep Díaz who kept a vigilant eye in my activities, facilitated financial support when necessary and showed concern and encouragement when I needed them most. Rafel Cases who was my *tutor* when I first arrived to the UPC and started me in the analysis of algorithms. And Vladimir Estivill-Castro who introduced me to computer science and to research.

I would also like to thank those who over the years have given me their friendship, but they are too numerous to mention all of them here. I do, however, want to express special thanks to a few whose presence, criticism and encouragement were directly involved with this dissertation. Carme and Ana who have always been so close to me deserve a particular mention. Following them are: Edelmira, Ma. Amelia, Fatos, Nicola, Miguel, Balqui, Adriana, Rosa, Ma. José, Ma. Luisa, Xavier, Javier, Jordi P., Jordi M., Juan Luis, Cristina, Christian, Cora, Vianey, Luz Elena, Elena, among too many others.

Finally, I want to thank my family. My parents above all, my brothers and in-laws and with a special word, Pablo and Ximena who loved me through the tough and wonderful experience of writing and delivering this thesis.

Funding. I received financial support from CONACyT (México) under grant number 89422 and travel support from the Future and Emergent Technologies Program of the EU under contract IST-1999-14186 (ALCOM-FT) and the Spanish Min. of Science and Technology project TIC2002-00190 (AEDRI II) to present parts of this dissertation.

CONTENTS

It is interesting to note that the human brain is much better at secondary key retrieval than computers are; in fact, people find it rather easy to recognize faces or melodies from only fragmentary information, while computers have barely been able to do this at all. Therefore it is not unlikely that a completely new approach to machine design will someday be discovered that solves the problem of secondary key retrieval once and for all, making this entire section obsolete.

—Donald E. Knuth: The Art of Computer Programming, Second Edition, Volume 3: Sorting and Searching (1999).

Part I

INTRODUCTION

This thesis is about the design and analysis of multidimensional data structures, a field that has been in constant growth in the last decade due to the increasing popularity of applications that naturally require it, like web searching, geographical information systems, image processing, or robotics, among others [43, 95].

By *multidimensional data* we understand data that is usually represented in some vector-based form. Consequently, *multidimensional data structures* are data structures specifically designed for the storage and management of multidimensional data. In the literature, multidimensional data structures are also referred to as multidimensional access methods, spatial access methods or spatial index structures [43].

Within multidimensional data structures it is important to distinguish between point data structures and spatial data structures. *Point* data structures are multidimensional data structures that maintain a collection of items or records, each holding a distinct K-dimensional key (which we may assume w.l.o.g. is a point in $[0,1]^K$). *Spatial* data structures store more complex spatial objects such as lines, polygons or higher dimensional polyhedra.

In this thesis we devote our attention to point data structures[1], not only because many applications involve them, but because it is not uncommon to represent more complex spatial data and even complex non-spatial objects, such as multimedia files or text documents, as multidimensional points generally called *surrogates*. Surrogates are frequently used in order to create indexes of complex data, in such a way that many of the required manipulations can be performed in simpler elements (the surrogates) and consequently become more efficient. Moreover, it is often the case that large data bases are stored in external memory while all the accesses are made through the indexes via the surrogates. In what follows we will use the term *multidimensional data structures* to refer to point multidimensional data structures, unless otherwise stated.

Retrieval (or search) in multidimensional data structures is usually known as *associative retrieval*. Besides insertions, deletions, and (exact) search, a multidimensional data structure should allow efficient answers to questions like which records of the data structure do fall within a given hyper-rectangle Q (orthogonal range search) or which is the closest record to some given point q according to some distance or similarity metric (nearest neighbor

[1] But many of the techniques we use in this work can be applied also to other kinds of multidimensional data structures.

search) [43, 95], to mention a few.

The design of multidimensional data structures must take into account the intrinsic properties of multidimensional data and their applications. Specifically, multidimensional data is dynamic, since applications involve series of interleaved insertions, deletions, and associative queries, multidimensional data tend to be large, the set of requirements depends strongly on the particular application, and in general, associative operations are more expensive than standard ones. As stated by V. Gaede and O. Günther [43],

> "*the challenge for the developers of a spatial database system lies not so much in providing yet another collection of special-purpose data structures. Rather, one has to find abstractions and architectures to implement generic systems, that is, to build systems with generic spatial data-management capabilities that can be tailored to the requirements of a particular application domain*".

With this goal in mind, we focused our attention in K-d trees [4] and quad trees [6], two basic general-purpose multidimensional data structures that are widely used in applications because they support a large set of associative queries with reasonable space and time requirements. One of the main drawbacks of these hierarchical structures is that their performance depends on the randomness of the inputs. Furthermore, the existing deletion algorithms do not preserve randomness, so that the overall performance of these data structures may degrade significantly after long sequences of interleaved insertions and deletions [21, 32].

In order to overcome these problems, we have proposed randomized K-d trees and randomized quad trees. These new data structures are fully dynamic (in the sense that they support insertions and deletions in any order with the guarantee that their expected time bounds hold), they efficiently support a broad range of associative queries, they are simple to describe and implement, and they work for any dimension.

We obtained these results through theoretical and experimental analysis of the proposed data structures. In particular, a noteworthy contribution of this thesis is our mathematical analysis of the performance of orthogonal range queries for randomized K-d trees, which is actually applicable to a broad collection of multidimensional data structures. In general, the average-case analysis of the performance of orthogonal range queries for multidimensional data structures has proven to be a difficult problem. The orig-

inal analysis for standard K-d trees by Bentley *et al.* and most subsequent work [4, 99] rely on the unrealistic assumption that the considered tree data structure is perfectly balanced, and thus, provide unduly optimistic results. Progress came with two papers [14, 24] that provide upper (Ω) and lower bounds (big-Oh) for the average performance of range search in standard K-d trees, squarish K-d trees, and other multidimensional data structures.

In this thesis, we analyze the average cost of range queries for a large class of hierarchical multidimensional data structures using the same random model as in [14, 24] but completely different techniques, and we obtain sharper results. In particular, we get exact upper and lower bounds and a characterization of the cost of range search as the sum of the cost of partial match-like searches. Using these results, we provide tight asymptotic estimates for the expected cost of range search.

These new data structures, and in particular randomized K-d trees, have attracted the attention of other researchers as witnessed by the references to our work in [14, 17, 58, 72, 78, 79].

In another line of work, we take advantage of locality of reference in multidimensional search. Although locality of reference is systematically used in the design of memory hierarchies (disk and memory caches) and it is the rationale for many other techniques like buffering and self-adjusment [11, 100], it was not exploited in the case of multidimensional data. We introduce a new type of data structure, the fingered multidimensional trees, which are easy to implement and yield significant savings under reasonable models of orthogonal range and nearest neighbor queries that exhibit locality of reference.

The thesis is organized as follows. Part I corresponds to this introduction. In Part II we give an overview of the field of multidimensional data structures and the notation and basic definitions to be used later on.

The design of multidimensional data structures is the subject addressed in Part III, which includes:

1. The design of *Randomized K-d Trees* (Chapter 3),
2. The design of *Randomized Quad Trees* (Chapter 4), and
3. The design of *Fingered Multidimensional Trees* (Chapter 5).

In Chapters 3 and 4 we introduce randomized K-d trees and randomized quad trees, variants of classical K-d trees [4] and quad trees [6], respec-

tively. Randomized K-d trees are a generalization of randomized binary search trees [73]. We first define *relaxed K-d trees*, K-d trees in which the sequence of discriminants in a path from the root to any leaf is arbitrary, contrary to standard K-d trees, where the sequence of discriminants along any path is cyclic, starting with the first coordinate. The flexibility of relaxed K-d trees (Chapter 3) allows us to use randomization to perform insertions and deletions in regions of the trees other than the leaves, in such a way that the resulting trees have all the properties of randomly built trees. Hence, the expected case performance of every operation holds, since it only depends on the random choices made by the randomized update algorithms.

Randomized quad trees, whose definition is similar to the one presented above, have other noteworthy advantages. We present and study them in Chapter 4. While deletion in standard quad trees [93] depends on a specific notion of distance and becomes more cumbersome as the spatial dimension grows, the randomized deletion algorithm for quad trees that we present here is defined for any dimension and is independent of any notion of proximity or distance between the stored spatial objects. The expected values of random variables (such as internal path length, depth, cost of successful or unsuccessful search, cost of partial match queries, among others) given in the literature [23, 25, 36, 70] for random quad trees are valid for randomized quad trees, because, as for randomized K-d trees, the randomized update algorithms produce random quad trees.

The third proposed data structures, which we introduce in Chapter 5, the fingered multidimensional trees, have been designed with on-line settings in mind, where requests arrive one at a time and they must be attended as soon as they arrive (or after some small delay). In such cases, we frequently encounter *locality of reference*, that is, for any time frame only a small number of different requests among the possible ones are made or consecutive requests are close to each other in some sense. The performance of searches and updates in data structures can be improved by augmenting the data structure with *fingers*, pointers to the hot spots in the data structure where most activity is going to be performed for a while (see for instance [12, 45]). Thus, successive searches and updates do not start from scratch but use the clues provided by the finger(s), so that when the request affects some item in the "vicinity" of the finger(s) it can be attended more efficiently.

In this work, we will concentrate in fingering techniques applied to two variants of K-dimensional trees, namely *standard K-d trees* [4] and *relaxed K-d trees*, but the techniques can easily be applied to other multidimen-

sional search trees and data structures. We propose two alternative designs (1-finger and m-finger K-d trees) that augment K-d trees with fingers to improve the efficiency of orthogonal range and nearest neighbor searches. We thereafter study their performance under reasonable models which exhibit locality of reference. While it seems difficult to improve the performance of multidimensional data structures using self-adjusting techniques (as reorganizations in this type of data structures is too expensive), fingering yields significant savings and is easy to implement. Our experiments show that the more complex scheme of m-finger K-d trees exploits better the locality of reference than the simpler 1-finger K-d trees; however these gains probably do not compensate for the amount of memory that m-finger K-d trees need, so that 1-finger K-d trees might be more attractive on practical grounds.

Part IV of the thesis is devoted to the analysis of associative queries in multidimensional data structures. In this case, using average-case analysis techniques we provide:

1. The analysis of partial match queries in random relaxed K-d trees (Chapter 7), and

2. The average-case analysis of orthogonal range search in a large class of hierarchical spatial data structures (Chapter 8).

Chapter 7 is devoted to the analysis of partial match queries. As shown by Flajolet and Puech [38], the expected time performance, measured as the number of visited nodes, for a partial match query in random K-d trees is $\Theta(n^{1-\frac{s}{K}+\theta(\frac{s}{K})})$, where n is the number of nodes in the tree, s is the number of specified attributes in the query, $0 < s < K$, and $\theta(x)$ is a real valued function whose magnitude never exceeds the value 0.07. In our work, we show that the expected number of visited nodes in a partial match query with s specified attributes in a random relaxed K-d tree of size n is $\beta n^{1-\frac{s}{K}+\phi(\frac{s}{K})}+\mathcal{O}(1)$, where we provide the closed forms for ϕ and β as functions of s/K; in particular the value of $\phi(x)$ never exceeds 0.12. The simplicity and flexibility of relaxed K-d trees is reflected also in a simpler and more complete mathematical analysis of partial match queries than those available for standard K-d trees and other variants [14, 17, 24, 38, 72, 79]. In particular, the exact value for β depends only on s/K, whereas for standard K-d trees the cost of the partial match is also dependent on the pattern structure of the query.

We address, in Chapter 8, the mathematical analysis of the performance of range searches. We analyze first the average cost of range search in randomized K-d trees. In our analysis we use the previous analysis of partial matches as a building block, together with combinatorial and geometric arguments, to provide a tight asymptotic estimate.

In particular, let $\mathbb{E}[R_n]$ be the expected cost of a range search with a random query in a random relaxed K-d tree of size n. Then we show that

$$\begin{aligned}\mathbb{E}[R_n] \;\sim\; & \Delta_0\Delta_1\cdots\Delta_{K-1}\cdot n + \sum_{0<j<K} c_j\cdot n^{1-j/K+\phi(j/K)} \\ & +2\cdot(1-\Delta_0)\cdots(1-\Delta_{K-1})\cdot\log n + \mathcal{O}(1),\end{aligned}$$

where Δ_r is the length of the range for coordinate r, and we give explicit values for the c_j's. Similar results hold for the rest of the multidimensional data structures mentioned above, but different c_j's and ϕ appear in each instance. This result can be used to show that, in random relaxed K-d trees, nearest neighbor queries are answered on-line in expected $\Theta(n^\rho+\log n)$ time, where $\rho=\max_{0<s<K}(\phi(s/K))$.

We finish the chapter by discussing how our results generalize to other multidimensional data structures, namely, standard K-d trees, squarish K-d trees, K-d-t trees, standard and relaxed K-d tries, quad trees and quad tries [4, 6, 13, 22, 24, 89].

Finally, in Part V, we review the main conclusions of the present work proposing a brief account of some open problems raised by our research.

Chapter 3 is based on *Randomized K-dimensional Binary Search Trees* [27]. Most of the material in Chapter 4 is presented in *Randomized Insertion and Deletion in Point Quad Trees* [26] and the results of Chapter 5 appear in *Fingered Multidimensional Search Trees* [31].

The material of Chapter 7 is presented in *Randomized K-dimensional Search Trees* [27]. Chapter 8 is based on the paper *On the Average Performance of Orthogonal Range Search in Multidimensional Data Structures* [29, 30].

Part II

PRELIMINARIES

1. THE ASSOCIATIVE RETRIEVAL PROBLEM

1.1 Multidimensional Data

Points, lines or hyper-planes lying in Euclidean space; strings of fixed length lying in Hamming space; strings of variable length lying in Leveshtein space; geographic maps, books or other text documents, acoustic or musical data, proteins, are all examples of multidimensional data. Multidimensional data objects are present in a broad range of applications in computer science such as databases, data-mining, computer graphics, robotics, medical images, neural networks, multimedia, statistical computing, computer-aided design, astronomy, pattern recognition, geographic information systems, music information retrieval, computational biology, etc. For information about these specific applications we refer the reader to [94] and the references therein.

In general, by *multidimensional data* we mean data that contain explicit information about objects, their extent, and/or their position in space. This information is usually represented by vectors. And it is generally assumed that multidimensional data objects lie in a K-dimensional space which is referred to as *universe*. Here K is a natural number corresponding to dimensionality.

An explicit example of two-dimensional geographic data is shown in Table 1.1 which lists some localities in Catalonia together with their corresponding latitude and longitude coordinates. Figure 1.1 shows the geographic localization of the listed localities.

Multidimensional data structures are multidimensional data management systems that support search and update operations in multidimensional data. In the literature, multidimensional data structures are also referred to as multidimensional access methods, spatial access methods or spatial index structures. Within multidimensional data structures it is important to distinguish between point multidimensional data structures and spatial multidimensional data structures. The *point* data structures are multidimensional data structures that store points in two or more dimensions. *Spatial* data structures can store extended spatial objects such as lines, polygons or higher dimensional polyhedra, among others.

Multidimensional data have basic properties that are essential in the study of their management systems. Specifically, spatial data have a complex structure (because they can consist of several polygons arbitrarily distributed in space), are dynamic (since applications involve series of interleaved insertions, deletions and associative queries) and tend to be large (geographic maps, for instance, can occupy some gigabytes of storage). Moreover, there

Locality	Longitude	Latitude
(A) Arenys de Mar	2°33′ E	41°33′ N
(B) Barcelona	2°11′ E	41°23′ N
(C) Cardona	1°49′ E	41°56′ N
(D) Delta de l'Ebre	0°45′ E	40°45′ N
(E) Empúries	3°15′ E	42°20′ N
(F) Figueres	2°58′ E	42°14′ N
(G) Girona	2°49′ E	41°59′ N
(H) Hostalric	2°45′ E	41°45′ N
(I) Igualada	1°37′ E	41°35′ N
(J) Jonquera	3°00′ E	42°30′ N
(L) Lleida	0°38′ E	41°37′ N
(M) Manlleu	2°17′ E	42°00′ N
(N) Nùria	2°13′ E	42°30′ N
(O) Olot	2°30′ E	42°11′ N
(P) Puigcerda	1°56′ E	42°26′ N
(Q) Querol	1°30′ E	41°30′ N
(R) Reus	1°06′ E	41°10′ N
(S) Seu d'Urgell	1°28′ E	42°22′ N
(T) Tarragona	1°16′ E	41°07′ N
(U) Ullastret	3°10′ E	42°16′ N
(V) Vic	2°15′ E	41°56′ N
(X) Xert	0°15′ E	40°44′ N

Tab. 1.1: A two-dimensional file of localities in Catalonia.

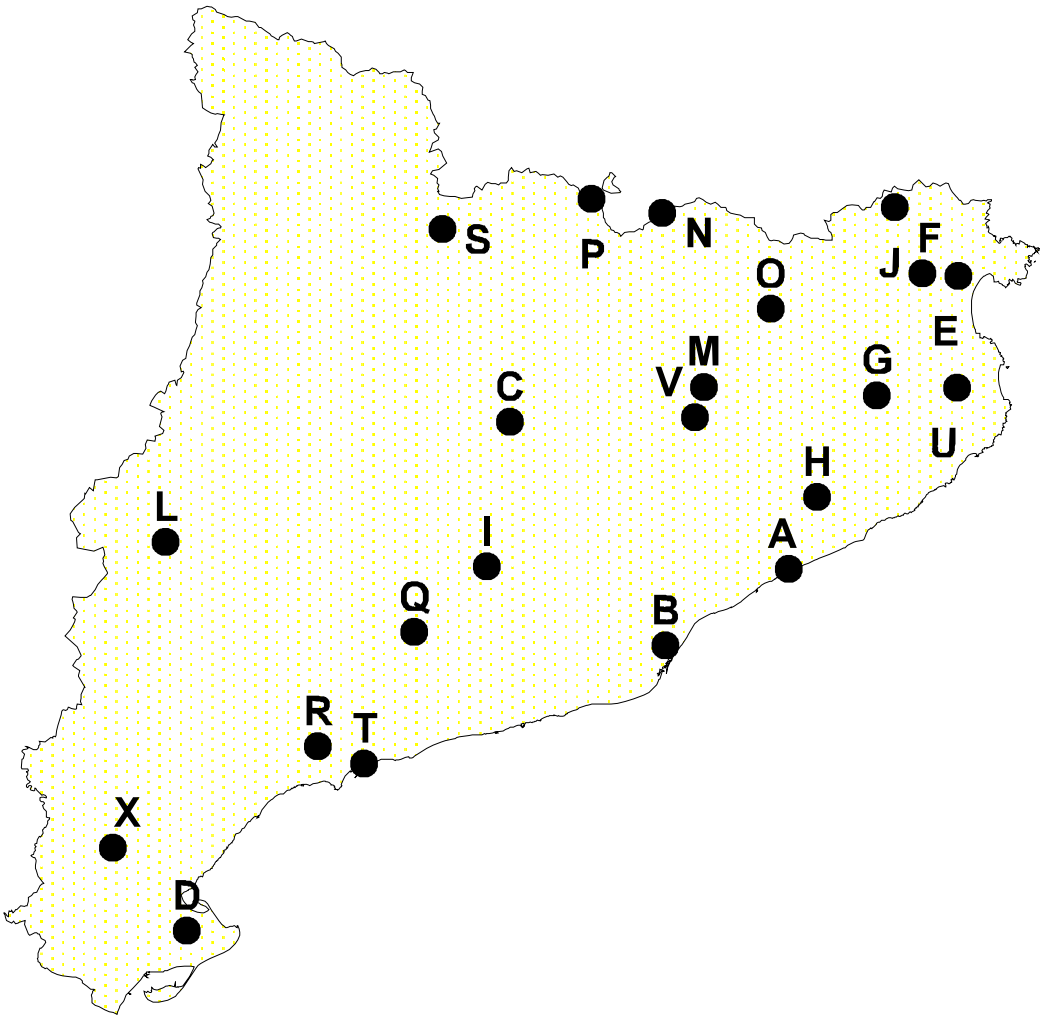


Fig. 1.1: Geographic representation of the Catalan localities listed in Table 1.1.

is no standard set of spatial operators (since they strongly depend on the particular application), many spatial operators are no closed (the intersection of two polygons, for instance, may not be a polygon) and in general, multidimensional operators are more complicated than standard ones.

The objects of study in this thesis are point multidimensional data structures. This is because multidimensional data points are frequently present in applications and also because it is common to represent extended data (or objects), like text documents or multidimensional files, as multidimensional points generally called *surrogates*.

A practical example of surrogates is the representation of characters in the 16-bit Unicode code space. For instance, the Chinese speaking community alone uses over 55,000 characters of the 65,536 that fit in the system (without surrogates). The Unicode Standard augments its character capacity by defining surrogates. In this case a surrogate is a pair of 16-bit Unicode code values that represent a single character. In this way, Unicode can support over one million characters. For more details we refer the reader to The Unicode Standard System [20], version 2.0.

There are two frequent ways for assigning spatial extended objects to multidimensional points. The first possibility is to transform each extended object into a higher multidimensional point [53, 97]. For example, two-dimensional rectangles can be represented as four dimensional points either taking the coordinates of two of their diagonal corners (end point transformation) or taking its centroid together with its extension in both coordinates (middle point transformation). The second option is to transform extended objects into a set of one-dimensional intervals by means of *space filling curves* [92]. There are several variations of the space filling curves technique. Examples are: z-ordering [85], the Hilbert tree [57], and the UB-tree [2].

Another important application of surrogates is that they are frequently used in order to create indexes of complex data. Indexes are then used to perform either simpler manipulations of the complex data objects or accesses to large multidimensional data bases stored in external memory.

In what follows, unless otherwise stated, we will use the term multidimensional data structures to refer to point multidimensional data structures. We also will look at points in a K-dimensional space as multidimensional records. For the sake of simplicity, we identify a multidimensional record with its corresponding K-dimensional key $x = (x_0, x_1, \ldots, x_{K-1})$, where each x_i, $0 \leq i \leq K-1$, refers to the value of the i-th attribute of the key x. As

an example, consider the file given in Table 1.1, consisting of some localities in Catalonia. Each record corresponds to one locality and has associated two attributes: the latitude and longitude of the locality; so $K = 2$ in this example. In general, each x_i belongs to some totally ordered domain D_i, and x is an element of $D = D_0 \times D_1 \times \ldots \times D_{K-1}$. Without loss of generality, we assume that $D_i = [0, 1]$ for all $0 \leq i \leq K - 1$, and hence that D is the hypercube $[0, 1]^K$ [70]. And the file F of n multidimensional records is simply $F = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$.

1.2 Associative Retrieval

Retrieval (or search) in multidimensional data structures is generally known as *associative retrieval* (also referred to as retrieval by secondary key, retrieval by composite key, or as multi-key retrieval). In our work associative retrieval consists of requesting and retrieving information from a collection of multidimensional points (or records). There is a condition imposed to requests, they must deal with more than one of the point coordinates. If this condition is not satisfied and all the requests deal with only one of the coordinates, then, one-dimensional data structures have better performance than multidimensional ones.

Associative queries are usually divided into two groups: intersection and proximity queries. Intersection queries, in turn, are classified in exact match queries, partial match queries and region queries.

An exact match query is a query in which all key attributes are specified. The retrieval consists of finding all records in the file with the given key. In the file of Table 1.1 an exact match query could be, for instance, a request for a record with attributes 2°15′ E of longitude and 41°56′ N of latitude (Vic). This is formally stated by the next definition.

Definition 1.2.1. Given a file F of n K-dimensional records and a K-dimensional query record q, an *exact match query* consists of retrieving the point x in F whose coordinates match the coordinates of q. This is,

$$\{x \in F \mid x_i = q_i, \forall i \in \{0, \ldots, K - 1\}\}.$$

A partial match query is a query in which only s out of the K attributes of a given key are specified (with $0 \leq s < K$). The retrieval consists of finding all the records whose specified key attributes coincide with those of

the given query. Retrieving in Table 1.1 all those records with latitude equal to 40°45′ N (Delta de l'Ebre) is an example of partial match query. The formal definition is as follows.

Definition 1.2.2. Given a file F of n K-dimensional records and a query $q = (q_0, q_1, \ldots, q_{K-1})$ where each q_i is either a value in D_i (it is specified) or $*$ (it is unspecified), a *partial match query* returns the subset of records x in F whose attributes coincide with the specified attributes of q. This is,

$$\{x \in F \mid q_i = * \text{ or } q_i = x_i,\ \forall i \in \{0, \ldots, K-1\}\}.$$

Given a bit-string w, a record $y \in D$, and a partial match query $q = (q_0, q_1, \ldots, q_{K-1})$, where $q_i = y_i$ if $w_i = 1$ and $q_i = *$ if $w_i = 0$, we say that w is the specification pattern of the partial match query q.

To every partial match query with record y and specification pattern w we associate the hyperplane $H(y, w)$ defined by

$$H(y, w) = \{x \in F \,|\, \forall i : w_i = 1 \implies x_i = y_i\}.$$

Notice that the value of y_i in the definition of $H(y, w)$ is irrelevant if $w_i = 0$.

For instance, if $K = 2$ and $D = [0, 1]^2$, then $H(y, 00) = D$, $H(y, 11) = \{y\}$, $H(y, 01)$ is a line passing through y parallel to the horizontal axis and $H(y, 10)$ is a line passing through y parallel to the vertical axis.

A region query is a more general type of query. In this case, the query defines any region of the K-dimensional space of keys. The retrieval consists of searching in the file for all the points that fall inside the given region. When the region query is a hyper-rectangle whose sides are orthogonal to the coordinate axis we say that it is a range query. As an example, consider a request for all those localities in Table 1.1 with longitude between 0°00′ E and 1°00′ E and latitude between 40°00′ N and 42°00′ N (Delta de l'Ebre, Lleida and Xert). Formally we have the following.

Definition 1.2.3. Given a file F of K-dimensional records and a hyper-rectangle $Q = [l_0, u_0] \times [l_1, u_1] \times \cdots \times [l_{K-1}, u_{K-1}]$, an *orthogonal range query* returns the subset of records in F which fall inside Q. This is,

$$\{x \in F \mid l_i \leq x_i \leq u_i, \forall i \in \{0, \ldots, K-1\}\}.$$

Therefore, exact match queries are region queries in which the regions are isolated points and partial match queries with s attributes specified correspond to the region being a $(K - s)$-dimensional hyper-plane.

A request for the closest record to a given query record, under a determined distance function, is called a nearest neighbor query. A nearest neighbor query is, for instance, a request for the locality in Table 1.1 closest to $(2°11', 41°23')$ (Barcelona). The definition is the following.

Definition 1.2.4. Given a file F of K-dimensional records and a query point q, a *nearest neighbor query* consists of finding the key in the file closest to q according to some predefined *distance* measure d. This is,

$$\{x \in F \mid d(q,x) \leq d(q,y), \forall y \in F\}.$$

Nearest neighbor queries can be extended to m-nearest neighbor queries: given a query point q, it is required to find the m keys in the file closest to the given one, according to a given distance measure.

Definition 1.2.5. Given a file F of K-dimensional records and a query point q, a *m-nearest neighbor query* consists of finding the m keys in F closest to q according to some predefined *distance* measure d. This is,

$$\{A \subseteq F \mid |A| = m \text{ and for all } x \in A,\ d(q,x) \leq d(q,y), \text{ for all } y \in F\}.$$

A request for the record(s) in the file at a distance at most δ from a given point is a proximity query.

Definition 1.2.6. Given a file F of K-dimensional records, a query point q, and a distance value δ, a *proximity query* consists of finding the keys in the file with distance at most δ from q according to some predefined distance measure d. This is,

$$\{x \in F \mid d(q,x) \leq \delta\}.$$

This type of query can also be seen as a region query, but it shares a common flavor with nearest neighbor queries.

2. HIERARCHICAL MULTIDIMENSIONAL DATA STRUCTURES

2.1 Basic Multidimensional Data Structures

Nowadays, numerous data structures and algorithms exist for handling multidimensional data and associative queries, and much work is going on in the subject [43, 60, 95]. There exist also several taxonomies for the classification of these data structures [95, 97]. In general, they can be separated into two main groups: the hierarchical multidimensional data structures (tree-like) and the non-hierarchical ones (mostly based on hash). Multidimensional data structures may reside either in main or in external memory. Since multidimensional data structures tend to be very large, it is common to store data in external multidimensional access structures and to use main memory structures as indexes to access them [43].

In this chapter, we focus our attention in main memory hierarchical multidimensional data structures not only because they are the basis of the contributions of this thesis but because they also are the basis of the development of the field. However, we mention the original data structures in which are based the classes of: (a) non-hierarchical data structures and (b) hierarchical data structures lying in external memory. We do not pretend to give an exhaustive description of the existing hierarchical data structures to handle multidimensional data, but to give a flavor of the existing techniques and the historical development of the field in order to provide a better understanding of the associative retrieval problem. The description herewith is very general and concentrates only on the main characteristics of each method. We focus mainly in those hierarchical data structures that had an impact on the design of multidimensional data structures.

In some cases, in order to simplify the description of multidimensional data structures and the algorithms governing them, it is assumed that each of the points to be stored is unique. In other words, that the K-dimensional records to be stored are all compatible.

Definition 2.1.1. We say that two K-dimensional keys x and y are *compatible* if, for every $i = 0, 1, \ldots, K-1$, their i-th attributes are different.

Note that any two keys drawn uniformly and independently from $[0,1]^K$ are compatible, since the probability that $x_i = y_i$ is zero.

If an application permits collisions (i.e., several data points with the same attribute values), then one possibility to handle this situation is to provide the data structure with an additional field in which a pointer to an overflow collision list would be stored. In such a case, the update algorithms should

Long.	$X \to L \to D \to R \to T \to S \to Q \to I \to C \to P \to B \to N \to V \to M \to O \to A \to H \to G \to F \to J \to U \to E$
Lat.	$X \to D \to T \to R \to B \to Q \to A \to I \to L \to H \to V \to C \to G \to M \to O \to F \to U \to E \to S \to P \to N \to J$

Tab. 2.1: Inverted file built from Table 1.1.

be slightly modified. It could be argued that devoting an extra field to an unfrequent event such as a collision wastes storage; however, without it, more complicated update procedures would be required. Another possibility to solve this problem is to determine rules for handle collisions. In the case of binary trees, for instance, it is usual to decide arbitrarily to give preference to one side of the tree, lets say right. That is, if a record x to be inserted is equal to a record y already in the tree, then x should be inserted always in the right subtree of the node containing y.

The assumption of compatibility of keys is usual in the average case analysis of the performance of associative queries and it will be required in Parts III and IV of this thesis.

The simplest and straightforward approach to associative retrieval is to store the records in a sequential *list*. As a query arrives, all the elements of the list are sequentially scanned and every record that satisfies the query is reported. If the queries do not have to be handled immediately, then, they can be batched so that many queries can be processed with a single sequential pass through the file. For a file of n K-dimensional records, the list structure has the following properties; $\Theta(Kn)$ time to build the list, $\Theta(Kn)$ storage is required and associative queries are answered in $\Theta(Kn)$ (worst-case) time. Lists have the advantage of being very easy to implement. They are competitive with the more sophisticated methods described in this work when the file is small and the number of attributes large, or when the application is such that a large number of file records satisfy the query.

A natural extension to the use of lists is the *projection* technique (also called *inverted files* [60]). It consists of keeping, for each attribute, a sorted sequence of the records in the file. Geometrically, this corresponds to projections of the points on each coordinate. The K lists representing the K projections can be obtained by using K times some standard sorting algorithm. The inverted file corresponding to the file of Table 1.1 of localities in Catalonia[1] is represented in Table 2.1.

[1] In what follows we will refer to localities in the file of Table 1.1 only by its leading character.

To preprocess a file of n K-dimensional records we must perform K sorts of n elements, which takes $\Theta(Kn \log n)$ time. To store such a file require $\Theta(Kn)$ space. An exact match query can be answered by searching (using binary search) any of the K sorted lists in $\Theta(\log n)$ worst-case time. The worst-case cost for any kind of associative queries is $\Theta(Kn)$. Range queries can be answered by the following procedure: choose one of the attributes, say the i-th. Find the two limits of the range query in the appropriate sorted sequence (using binary search). All the records satisfying the query will be in the list between these two positions. This (usually smaller) list is then searched by brute force. For almost cubical range queries that have a small number of records satisfying them (and are therefore similar to nearest-neighbor searches), the range query time of projection is given by $\Theta(n^{1-\frac{1}{K}})$ in the average case when the point set is drawn from a smooth underlying distribution [8]. In solving range queries, the projection technique is most effective for queries containing one range that excludes most of the records in the file. If the distribution of attribute values is more or less uniform over similar ranges and the query range is cubical, then we can use the projection technique with only one sorted list.

This technique has been applied by Friedman, Baskett, and Shustek [41] in their algorithm for nearest-neighbor queries, in particular they show that nearest-neighbor queries can be answered using projection in $\Theta(n^{1-\frac{1}{K}})$ time and $\Theta(Kn \log n)$ preprocessing. Also Lee, Chin and Chang have applied projection to a number of database problems [67].

The *cell* or *fixed grid* method, first introduced by Nievergelt, Hinterberger and Sevcik [80] and based in extendible hashing, divides the space into equal sized cells or buckets (squares and cubes for two and three dimensional data respectively). The data structure is a K-dimensional array with one entry per cell. Each cell is implemented as a linked list storing the points within it. In Figure 2.1 we illustrate one possible grid for Table 1.1.

The space required for this data structure can be super-linear in the number n of records, even if the data is uniformly distributed. In particular, Regnier [88] showed that the average size of the grid file while storing n K-dimensional records is $\Theta\left(n^{1+(K-1)/(Kb+1)}\right)$, where b is the bucket size, and that the average occupancy of the data buckets is approximately 69%. The analysis of the performance of grid files on associative queries [88] must take two costs into account; the number of required cell accesses (or number of directory look-ups) and the number of required inclusion tests (testing

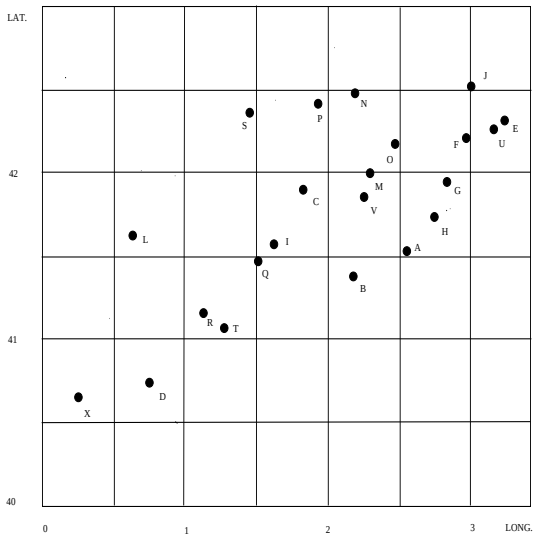


Fig. 2.1: Illustration of the cell method; a grid for the file in Table 1.1.

whether a point satisfies the query). If the cell size is large there will be few cell accesses and many inclusion tests. By contrast, if the cell size is small, there will be many cell accesses and few inclusion tests. Both extremes are unsuitable.

Range queries with constant size can be answered (in a file organized by cells having the same size than the queries) with approximately 2^K cell accesses [8]. The expected search time in this case is proportional to 2^K times the average number of points per cell. In such files nearest neighbor queries have similar performance. In most applications, however, the queries will vary in size and shape, and there is little information available for making a good choice of cell size (and possibly shape).

Most of the non-hierarchical multidimensional data structures are based, inspired or closely related in some way to grid files. Examples of such data structures are: *excell* [102] (extendible cell), *two-level grid file* [53], *twin grid file* [54], *molhpe* [61] (multidimensional order-preserving linear hashing with partial expansions), *quantile* hashing [62, 64], and *plop* [63] (piecewise linear order-preserving hashing), among others. We do not go further in the description of such data structures. Our interest in grid files is due to its relation with some of the hierarchical data structures that we explain below.

2.2 Data Structures Based on K-d Trees

Bentley [4] introduced multi-dimensional binary search trees (K-d trees) as a generalization of binary search trees. A K-d tree is a combination of cells and binary search trees in which the space is also divided into hyper-rectangles, but this time depending on the records' attributes.

Definition 2.2.1. A K-d tree for a set $F = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ of K-dimensional records is a binary tree such that:

1. Each node contains a K-dimensional record and has an associated discriminant $j \in \{0, 1, \ldots, K-1\}$.

2. For every node with key x and discriminant j, the following invariant is true: any record in the left subtree with key y satisfies $y_j < x_j$, and any record in the right subtree with key y satisfies $y_j \geq x_j$.

3. The root node has depth 0 and discriminant 0. All nodes at depth d have discriminant (d mod K).

Note that we do not need to explicitly store a field containing the discriminant of each node, since they are implicitly given by the third condition of the definition above.

We say that a node contains (x, j) if its key is x and its discriminant is j. Let us now give a definition that will be required later on.

Definition 2.2.2. We say that a K-dimensional key x is *compatible* with a tree T if, $\forall j \in \{0, \ldots, K-1\}$, its j-th attribute is different from the j-th attribute of the keys already in T.

A K-d tree for a file F can be incrementally built by successive insertions into an initially empty K-d tree as follows. If the tree T is empty, the insertion of x results in a K-d tree with root node x and empty subtrees. The first attribute of the second key is then compared with the first attribute of the key at the root: if it is smaller, the second key is recursively inserted in the (empty) left subtree; otherwise, it is recursively inserted in the (empty) right subtree. In general, when inserting a key x, we compare the key to be inserted with some key y at the root of some subtree: if y is at level j, then we compare $x_{(j \bmod K)}$ and $y_{(j \bmod K)}$, and recursively continue the insertion in the left or the right subtree of y, until a leaf (empty subtree) is found. In

Figure 2.2 we show the 2-d tree that results from the insertion of the keys of the file in Table 1.1 into an initially empty tree, in the same order as they have been listed. The figure also shows the partition of the space induced by the 2-d tree. It should be clear that if the same points were inserted in a different order they could yield a substantially different tree.

In discussing deletions, it is sufficient to consider the problem of deleting the root node of a (sub)tree. If the root to be deleted (say y) has no subtrees, then the resulting tree is the empty tree. If y does have descendants and discriminant j (with $0 \leq j \leq K-1$), then it must be replaced by either the node in its left subtree with greater j-th attribute or the node in its right subtree with smaller j-th attribute.

The standard model for the probabilistic analysis of K-d trees is that a *random* K-d tree of size n is built by inserting n points independently drawn from some continuous probability distribution defined over $[0,1]^K$. This is equivalent to assume that the probability that the $(n+1)$-th insertion fails in a given leaf of a random K-d tree of size n is the same for any of its $n+1$ leaves.

The expected cost of a single insertion in a random K-d tree is $\Theta(\log n)$ time while the expected cost of building the whole tree is $\Theta(n \log n)$ [4]. The worst-case cost for insertions in a K-d tree is $\Theta(n)$ while the worst-case cost of building a K-d tree of n nodes is $\Theta(n^2)$. The cost of deleting the root of a random K-d tree with n nodes is $\Theta(n^{1-\frac{1}{K}})$. But in the average, deletions have expected cost $\Theta(\log n)$ since the expected depth of the node to be deleted is logarithmic and the subtree beneath it has expected size $2 \log n$ [4, 71].

Exact match queries in random K-d trees obviously operate by following a path down the tree, exactly in the same way as if we were inserting that key: either we find it and report success or we reach a leaf, reporting failure. Therefore, they also require expected $\Theta(\log n)$ time and $\Theta(n)$ worst-case time.

The algorithm for partial match searches over K-d trees explores the tree in the following way. At each node of the K-d tree it examines the corresponding discriminant. If that discriminant is specified in the query then the algorithm recursively follows in the appropriate subtree, depending on the result of the comparison between the attribute of the query and the attribute of the key at the current node. Otherwise, the algorithm recursively follows the two subtrees. It has been shown by Flajolet and Puech [38] that partial match queries are efficiently supported by random K-d trees. The expected

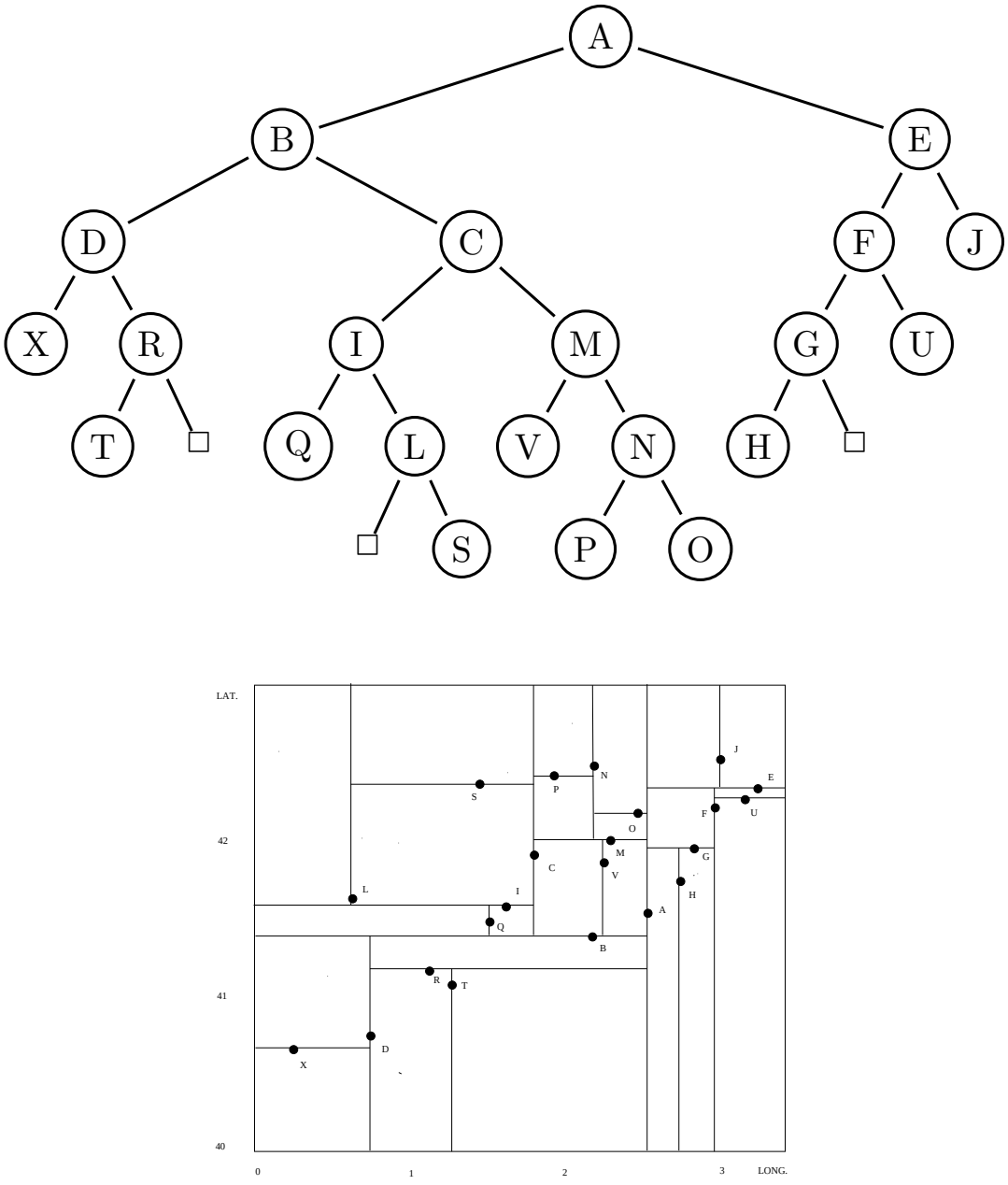


Fig. 2.2: Graphic and geometric representation of a K-d tree built from Table 1.1.

time of this operation in random K-d trees of size n is $\Theta(n^{1-\frac{s}{K}+\theta(\frac{s}{K})})$, where s is the number of specified attributes, $0 < s < K$, and $\theta(x)$ is a real valued function whose magnitude never exceeds the value 0.07.

The algorithm for orthogonal range queries is similar to the previous one. When visiting a node x that discriminates w.r.t. the j-th coordinate, we must compare x_j with the j-th range $[\ell_j, u_j]$ of the query. If the query range is totally above (or below) that value, we must search only the right subtree (respectively, left) of that node. If, by contrast, $\ell_j \leq x_j \leq u_j$, then both subtrees must be searched; additionally, we must check whether x falls or not inside the query hyper-rectangle. This procedure continues recursively until empty subtrees are reached. The pseudo-code is presented in Algorithm 1. From now on, we use the notation $p \rightarrow field$ to refer to the field *field* in the node pointed to by p. Usually, a K-d tree is represented by a pointer to its root node, and each node has three fields: key, left subtree and right subtree. In other variants of K-d trees (in the chapters to follow) it is also required a pointer to that node's discriminant since there is no implicit way for obtaining them.

Algorithm 1 The orthogonal range search algorithm for K-d trees.

```
function range_search(T : K-d tree, Q : query) : set
    if (T = nil) then return ∅;
    x := T → key;
    if (Q.u[j] < x[j]) then return range_search(T → left, Q);
    if (Q.l[j] ≥ x[j]) then return range_search(T → right, Q);
    set S := range_search(T → left, Q) ∪ range_search(T → right, Q);
    if (x ∈ Q) then S := S ∪ {x};
    return S;
end
```

One important concept related to orthogonal range searches is that of *bounding box* or *bounding hyper-rectangle* of a data point. In the case of K-d trees constructed with keys taken from $[0,1]^K$, a bounding hyper-rectangle is defined as follows.

Definition 2.2.3. Given an item x in a K-d tree T, its *bounding hyper-rectangle* $B(x) = [\ell_0(x), u_0(x)] \times \ldots \times [\ell_{K-1}(x), u_{K-1}(x)]$ is the region of $[0,1]^K$ defined as follows:

1. If x is the root of T then $B(x) = [0,1]^K$;

2. If $y = (y_0, \ldots, y_{K-1})$ is the parent of x and it discriminates w.r.t. the j-th coordinate then,

   - if $x_j < y_j$, then $B(x) = [\ell_0(y), u_0(y)] \times \ldots \times [\ell_j(y), y_j] \times \ldots \times [\ell_{K-1}(y), u_{K-1}(y)]$;
   - if $x_j \geq y_j$ then $B(x) = [\ell_0(y), u_0(y)] \times \ldots \times [y_j, u_j(y)] \times \ldots \times [\ell_{K-1}(y), u_{K-1}(y)]$.

It can be observed that $B(x)$ is the region of $[0,1]^K$ that corresponds to the leaf replaced by x when it was inserted in the K-d tree. Taking the tree and the partition of Figure 2.2, it is easy to see that the bounding box of the node labelled A is $B(A) = [0°00', 3°30'] \times [40°00', 43°00']$. Node B has bounding box $B(B) = [0°00', 2°33'] \times [40°00', 43°00']$, while the bounding hyper-rectangle of node E is $B(E) = [0°00', 2°33'] \times [040°00', 43°00']$. The bounding hyper-rectangles of the rest of nodes can be recursively obtained.

There are several variants for nearest neighbor searching in K-d trees. One of them, which we will use in Chapter 5, works as follows. The initial closest point is the root of the tree. Then we traverse the tree as if we were inserting q. When visiting a node x that discriminates w.r.t. the j-th coordinate, we must compare q_j with x_j. If q_j is smaller than x_j we follow the left subtree, otherwise we follow the right one. At each step, we must check whether x is closer or not to q than the closest point seen so far, and update the candidate nearest neighbor accordingly. The procedure continues recursively until empty subtrees are reached. If the hyper-sphere, say B_q, defined by the query q and the candidate closest point is totally enclosed within the bounding boxes of the visited nodes then the search is finished. Otherwise, we must visit recursively the subtrees corresponding to those nodes whose bounding boxes intersect but do not enclose B_q. This is Algorithm 2 described below, where $\texttt{dist}(x, y)$ is a predefined distance function between the keys.

It is important to note that the algorithms for partial match, orthogonal range and nearest neighbor searches are essentially the same for any variant of K-d trees and quad trees (except for finger multidimensional trees, introduced in Chapter 5). The difference lies in the costs of the algorithms, which are dependent on the specific characteristics of the data structure.

The efficient expected performance of a K-d tree holds only under the assumption that it is random. Unfortunately, in practical applications, this assumption does not always hold. For instance, it fails when the keys to be

Algorithm 2 Nearest neighbor search algorithm for K-d trees.

```
▷ Precondition: T is not empty
▷ Initial call: NN(T, q, ∞, nn)
procedure NN(T : K-d tree, q : query, min_dist : distance value, nn : key) : void
    x := T → key;
    d := dist(q, x);
    if (d < min_dist) then
      min_dist := d;
      nn := x;
    if (q[j] < x[j]) then
      NN(T → left, q, min_dist, nn);
      other := T → right;
    else
      NN(T → right, q, min_dist, nn);
      other := T → left;
    if (q[j] − min_dist ≤ x[j] and q[j] + min_dist ≥ x[j]) then
      NN(other, q, min_dist, nn);
end
```

inserted are sorted or nearly sorted with respect to one of the attributes. Moreover, an alternation of deletions and insertions over a random K-d tree destroys its randomness [21, 32]. This happens even if every item in the file is equally likely to be deleted. After some updates (insertions and deletions), the tree may need to be rebuilt to preserve its efficient expected performance.

One possibility to overcome this problem is the use of optimized K-d trees [40, 101], assuming that the file of records is given a priori. Such K-d trees are perfectly balanced and their logarithmic depth is guaranteed. However, when insertions and deletions are to be performed, a reorganization of the whole tree is required. Thus, this alternative is not suitable unless updates occur rarely and most records in the file are known in advance, conditions that are not met in many practical situations.

Another approach is to introduce explicit constraints on the balancing of the trees, as in dynamically balanced K-d trees, in divided K-d trees or in K-d-t trees [22, 103, 86]. At each step, all update operations check whether a predefined balance criterion remains true after the insertion or the deletion of an element. If the balance constraint is violated then a rather complex reorganization of the tree is performed. Although these methods yield theoretical efficient searches (exact search and associative queries), and in some cases provide worst-case guarantees, they sacrifice the simplicity of

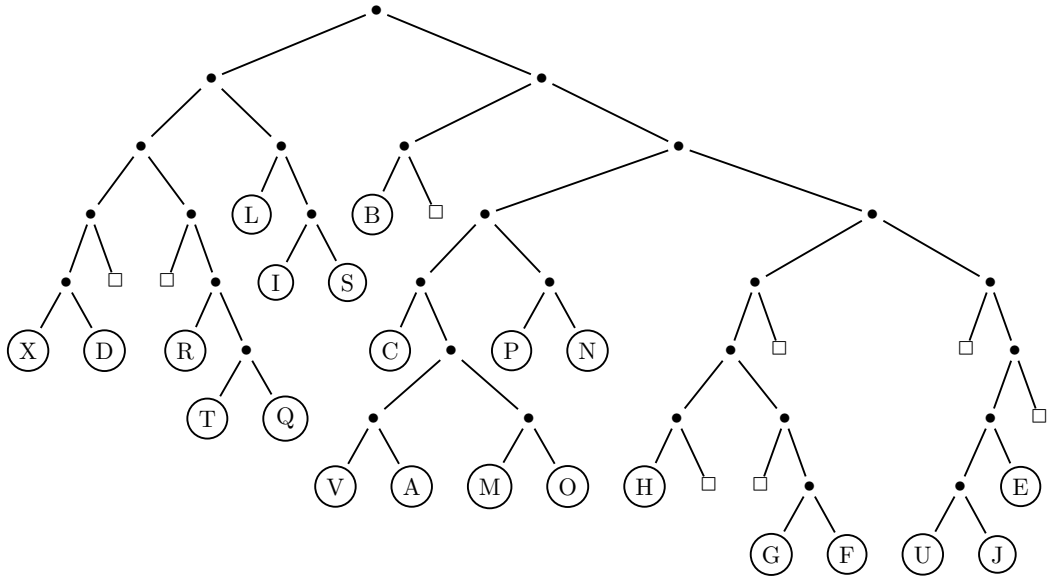
the standard K-d tree update algorithms and might be impractical in highly dynamic environments.

An interesting variant of K-d trees are squarish K-d trees proposed by Devroye, Jabbour and Zamora-Cura [24]. Assuming that the file of records is known a priori, squarish K-d trees are built by choosing as discriminant at each level the attribute with maximum spread in values, instead of assigning discriminants sequentially trough all the coordinates. The average cost of partial match queries in squarish K-d trees of n nodes is similar to the one of perfectly balanced complete K-d trees, namely $\Theta(n^{1-\frac{s}{K}})$, when s out of K attributes are specified. Despite the good performance of squarish K-d trees in partial match and other associative queries, they are unsuitable in dynamic environments because insertions and deletions may force a complete reorganization of the tree.

The idea of K-d trees can easily be generalized to digital search trees [13, 89] making a regular partition of the search space based on digits. The recursive partition of a region of the search space terminates when the region contains one (or no) data points. Searching in a binary K-d trie is as follows. At level 0 we use the first digit of the first key. If it is a zero the search proceeds to the left of the trie, and the search proceeds to the right if the bit is a one. The first bit of the second key is used in level 1, and so on, up to level $K-1$. Then, in level K we use the second bit of the first key and so on. In general, in level j we must use the $(\lfloor j/K \rfloor + 1)$-th bit of the $(j \mod K)$-th attribute of the given key. In Figure 2.3 we depict a 2-d trie for the set of records of the file in Table 1.1 and the induced partition of the search space. Note that both, the tree and the partition, depend on the given set of keys but not in the order in which they were inserted in the trie.

The probability model for the average-case analysis of K-d tries (and other variants) is usually the Bernoulli model, in which each of the K attributes of a key x is an infinite string of 0's and 1's, each bit independently generated, with identical probability (symmetric Bernoulli model) or with fixed probabilities p and $1-p$ (asymmetric Bernoulli model). The performance of partial match queries in K-d tries and other multidimensional digital structures (such as digital search trees or Patricia tries) is on the average $\Theta\left(n^{1-\phi}\right)$, where the coefficient, instead of being constant as in K-d trees, is a fluctuating function of the main order term [72].

We have just described K-d tree based multidimensional data structures for main memory storage, which are the basis of a wide range of data struc-



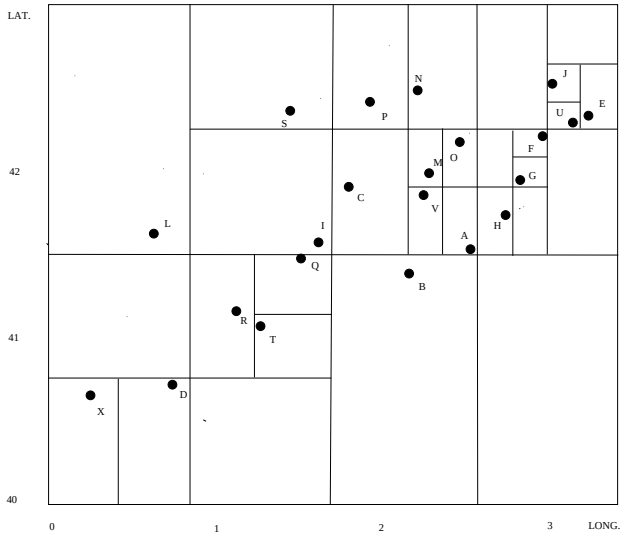


Fig. 2.3: Graphic and geometric representation of a K-d trie built from the file in Table 1.1.

tures for external memory. Let us mention for instance: KDB trees [90], Extended KD trees [74], BSP trees [42], BD trees [81], SKD trees [83], LSD trees [50], hB trees [69], GDB trees [82], G-trees [65], hB^{π} trees [33], and BV trees [39].

2.3 Data Structures Based on Quad Trees

Quad trees, introduced by Finkel and Bentley [34, 95], are also a generalization of binary search trees. In a 2-dimensional space the quad tree is given by the next definition.

Definition 2.3.1. A quad tree for a file $F = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ of 2-dimensional records is a quaternary tree in which:

1. Each node from F contains a 2-dimensional key and has associated four subtrees corresponding to the quadrants NW, NE, SE and SW.

2. For every node with key x the following invariant is true: any record in the NW subtree with key y satisfies $y_1 < x_1$ and $y_2 \geq x_2$; any record in the NE subtree with key y satisfies $y_1 \geq x_1$ and $y_2 \geq x_2$; any record in the SE subtree with key y satisfies $y_1 \geq x_1$ and $y_2 < x_2$; and, any record in the SW subtree with key y satisfies $y_1 < x_1$ and $y_2 < x_2$.

This definition for two-dimensional quad trees is readily generalized to an arbitrary dimension K (see Definition 4.1.1 in Chapter 4). The corresponding quad trees will have branching factor 2^K. For simplicity, most of the operations and properties that we describe in what follows are referred to 2-dimensional quad-trees, but they are easily generalized to higher dimensions.

A quad tree for a file F can be incrementally built by successive insertions into an initially empty quad tree as follows. The first record in the file is inserted as the root node. The insertion of the second record consists of comparing its attributes against the attributes of the root, in order to determine to which subtree it corresponds. If the appropriate subtree is empty then the record is inserted at its root. Otherwise the record is inserted recursively in the appropriate subtree. The remaining records are inserted in the same way. In Figure 2.4 we show the 2-d tree that results from the insertion of the keys of the file in Table 1.1 into an initially empty tree, in the same order as they have been listed. The figure also shows the partition

induced by the quad tree. It should be clear that if the same points were inserted in a different order they could yield a substantially different tree.

Deletion of nodes into two-dimensional quad trees is complicated. Finkel and Bentley [34] suggested that all nodes of the tree rooted at the deleted node must be reinserted, but this is usually expensive. A more efficient process developed by Sammet [95, 93] allows to reduce the number of nodes to be reinserted, although it is still an expensive and not straightforward process (it is described in Chapter 4).

Algorithms for exact search, partial match, orthogonal range search and nearest neighbor search are similar to those for K-d trees already described.

The standard model for the probabilistic analysis of quad trees is that a *random* quad tree of size n is built by inserting n points independently drawn from some continuous probability distribution defined over $[0,1]^K$.

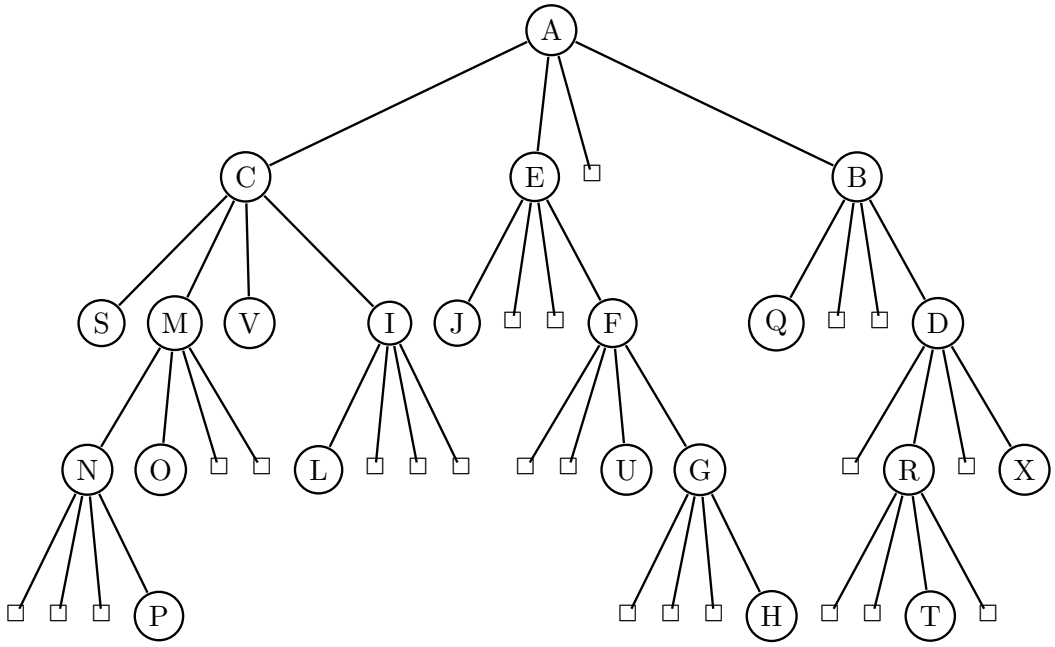
The expected height H_n of a K-dimensional quad tree of size n is in probability asymptotic to $(c/K)\log n$, where $c = 4.31107\ldots$ [23]. It has been shown independently by Devroye and Laforest [25] and Flajolet et al. [35] that the expected cost of a random search in a random K-dimensional quad tree of size $n-1$ is $(2/K)\log n$. For two-dimensional random quad trees of size $n-1$ Devroye and Laforest [25] show that the variance of a random search is $\sqrt{\frac{2}{K^2}\log n}$. A complete characterization of random searches in K-dimensional quad trees is given by Flajolet and Lafforgue [36]. In particular, they show that the cost of a random search has logarithmic mean and variance, and that it is asymptotically distributed as a normal variable, although no closed forms are given for $K \geq 3$.

The cost of a random partial match query with s coordinates specified in random quad trees of size n and dimension K is $\Theta(n^{1-s/K+\theta(s/K)})$, where the function $\theta(x)$ is defined as the solution $\theta \in [0,1]$ of the equation $(\theta+3-x)^x(\theta+2-x)^{1-x}-2=0$ [35].

The range search cost in quad trees has been studied by Bentley, Stanat and Williams [7] and by Lee and Wong [66]. In particular, Lee and Wong show that in the worst-case, range searching in a complete two-dimensional quad tree of size n takes $\Theta(n^{1/2})$ time. This result can be extended to K dimensions to yield $\Theta(n^{1-1/K})$ cost.

As it was the case for K-d trees, the efficient expected case performance of quad trees is based on the assumption that the quad tree is random. But this assumption does not always hold in applications.

Assuming that all nodes are known a priori, Finkel and Bentley [34]



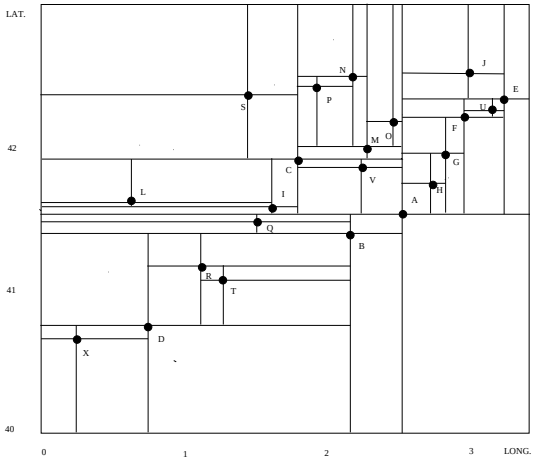


Fig. 2.4: Graphic and geometric representation of the quad tree built from the file in Table 1.1.

proposed an optimized quad tree such that given a node x, no subtree of x can contain more than one half of the nodes in the tree rooted at x. For instance, building an optimized two-dimensional quad tree requires sorting the records in the file by one of the attributes. The median value of the sorted file becomes the root of the quad tree, and the remaining records are regrouped into four sub-collections that will form the four subtrees of x. The process continues recursively in each subtree. This technique works because all the records preceding x in the sorted list will lie in the NW and SW quadrants (assuming that the first attribute serves as the primary key) and all the records following x will lie in the NE and SE quadrants. However, this method does not guarantee that the resulting tree will be complete.

Overmars and van Leeuwen [86] discuss a dynamic alternative approach, that is, the optimized quad tree is built during the insertion of records. The algorithm is similar to the previous one, except that every time the tree fails to meet a predefined balance criterion, it is partially re-balanced. Since one of the reasons that causes deletion in quad trees to be complex is that the data points serve to partition the search space, this pseudo quad tree simplifies deletion partitioning by arbitrary points not in the file. Overmars and van Leeuwen [86] show also that for any file of n records in a K-dimensional space, there exists a partitioning point such that every quadrant contains at most $\lceil n/(K+1) \rceil$ data points. They also demonstrate that the resulting pseudo quad tree has depth of at most $\lceil \log_{K+1} n \rceil$ and can be built in $\Theta(n \log_{K+1} n)$ time.

Although these methods yield theoretical efficient searches (exact search and associative queries), and in some cases provide worst-case guarantees, they might be impractical in highly dynamic environments.

Quad trees can be extended to digital settings in a similar manner than K-d trees. The resulting structure is called *quad trie*. For more details the reader is referred to [13, 72].

2.4 Other Hierarchical Multidimensional Data Structures

The contributions presented in this thesis are based in K-d trees and quad trees, but for completion other hierarchical multidimensional data structures are described herewith, specifically range trees and R-trees, with theoretical and/or practical importance and with an impact in the design of a wide class of multidimensional data structures.

Range trees were introduced by Bentley [5]. They achieve the best worst-case search time for range search among all the structures described so far, but they have large preprocessing and storage cost. For most applications, the high storage required by range trees is prohibitive, but they are still interesting from a theoretical point of view.

Range trees are recursively defined in dimension: the K-dimensional structure is defined in terms of the $(K-1)$-dimensional one. In other words, a *range tree* for a set of one-dimensional records is a sorted list where the elements are stored by key ascending order. A two-dimensional range tree is a rooted binary tree in which every node has associated a sorted array (one-dimensional range tree), a discriminant, and pointers to its left and right subtrees. The discriminant of every node is the median value of its records (sorted with respect to the first attribute), while arrays are sorted by the second attribute. The sorted array of the root contains all the nodes in the file. The root of its left subtree has a sorted array containing the records with first attribute smaller than the root's discriminant. Similarly, the right child represents the records with first attribute greater than the discriminant. This partitioning process continues until arrays consist of a single element.

In Figure 2.5 we depict the two-dimensional range tree that results from the insertion of the keys of the file in Table 1.1. In this example, every node contains its discriminant value taken from the first attribute (longitude) and its associated list sorted with respect to the second attribute (latitude).

To answer range queries in one-dimensional range trees we have to perform two binary searches over the list, in order to find the smallest and the greatest records that fall inside the range query. All the points in the array that lie between these two positions fall inside the range query and must be reported.

Range searches in two-dimensional range trees are described recursively as follows. Each node in the data structure represents a range in the first dimension going from the smaller first attribute contained in the subtree to the greatest. When visiting a node, we compare the range of the first attribute of the query against the range of the node. If the range of the node is entirely within the range of the query, then we search the sorted list of that node for all the points satisfying the query, and we list the points found. Otherwise, we compare the range of the first attribute of the query to the discriminant of that node. If the range is entirely below the discriminant, then we recursively visit the left subtree; if it is entirely above, then, we visit the right subtree; and if it overlaps the discriminant, then, we visit both

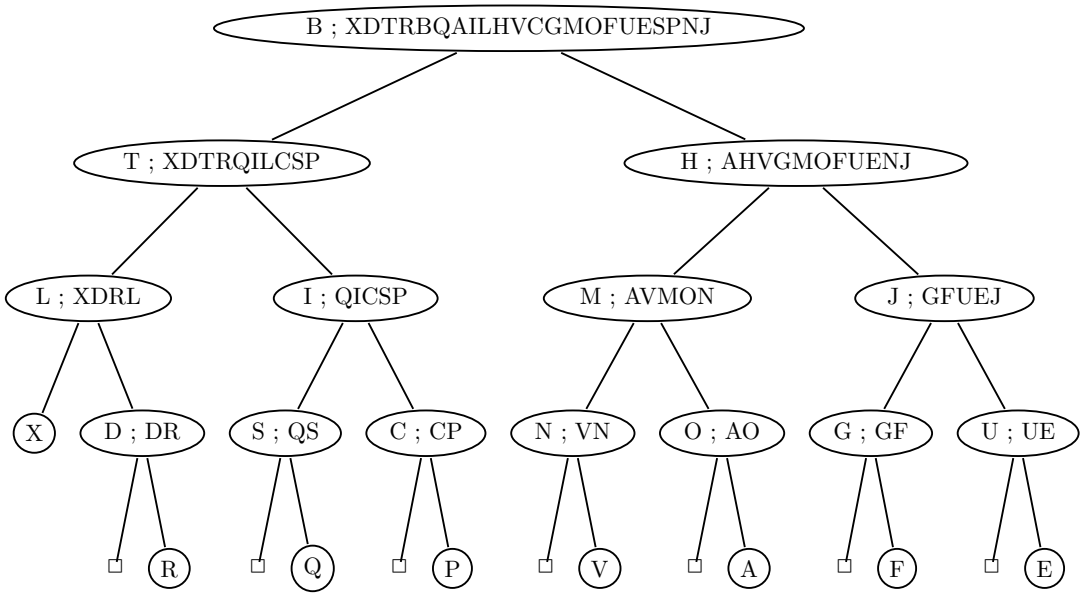


Fig. 2.5: Graphic representation of a range tree built from the file in Table 1.1.

subtrees recursively.

We constructed a two-dimensional range tree by building a tree of one-dimensional structures. We can perform essentially the same operation to obtain a three-dimensional structure; we build a tree containing two-dimensional structures in the nodes. This process can be continued to higher dimensions. The result is a K-dimensional range tree with $\Theta(n \log^{K-1} n)$ storage, $\Theta(n \log^{K-1} n)$ construction time and $\Theta(\log^K n)$ range search cost [8].

The data structures presented previously do not take into account paging in secondary storage. Now we describe R-trees, a tree structure designed for paged memory introduced by Guttman [48]. They are height-balanced trees based on B-trees [19]. Nodes correspond to disk pages if the index is disk-resident. Originally, R-trees were designed to deal with records identified with objects of non zero size, instead of being identified with K-dimensional points. They were generalized by Henrich and Moller [49] to handle multi-attribute records as well.

As B-trees, R-trees have a maximum number M and a minimum number $m \leq \frac{M}{2}$ of entries that will fit in one node. The leaf nodes of R-trees contain the record objects, while non-leaf nodes contain pointers to lower nodes in the tree. All leaf nodes appear in the same level of the tree.

Structure	Construction	Storage	Range Querie	Nearest Neighbor
List	$\Theta(Kn)$	$\Theta(Kn)$	$\Theta(Kn)$	$\Theta(Kn)$
Projection	$\Theta(Kn \log n)$	$\Theta(Kn)$	$\Theta(R + n^{1-\frac{1}{K}})$ (av.)	$\Theta(Kn)$
Cell	$\Theta(n)$	$\Theta(n)$	$\Theta(2^K F)$ (av.)	$\Theta(2^K F)$ (av.)
K-d tree	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(R + n^{1-\frac{s}{K}+\theta(\frac{s}{K})})$ (av.)	$\Theta\left(n^\rho + \log n\right)$ (av.)
quad tree	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(R + n^{1-\frac{s}{K}+\theta(\frac{s}{K})})$ (av.)	$\Theta\left(n^\rho + \log n\right)$ (av.)
range tree	$\Theta(n \log^{K-1} n)$	$\Theta(n \log^{K-1} n)$	$\Theta(\log^K n)$	

Tab. 2.2: Performance of some multi-dimensional data structures (F denotes the average number of records per cell, R is the number of points within the range, and av. indicates average cost).

Insertions in R-trees are similar to insertions in B-trees. New index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree. The search algorithm descends the tree from the root down in a similar way to B-trees. However, more than one subtree under a visited node may need to be searched, so it is not possible to guarantee good worst-case performance.

Deletions are also performed in a similar way to B-trees. First the record to be deleted must be located in a leaf of the R-tree. Then it must be removed, and finally, if the leaf-node has less than m entries after deletion, we must eliminate it and relocate its entries. If, during this re-accommodation, the root node has only one child, then its child must become the new root.

The height of a R-tree containing n records is at most $\lfloor \log_m n \rfloor - 1$, because the branching factor of each node is at least m. The maximum number of nodes is $n(\frac{1}{m} + \frac{1}{m^2} + \ldots) + 1$. The worst-case space utilization for all nodes except the root is $\frac{m}{M}$. Nodes tend to have more than m entries and this decreases the height of the tree and improves its space utilization.

There are several multidimensional data structures based on R-trees. Some examples are: packed R-trees [91], R^+-trees [98], R^*-trees [3], sphere trees [84], TR^* trees [96], parallel R-trees [56], P-trees [96], TV trees [68], Hilbert R-trees [57] and X-trees [9]. Examples of multidimensional data structures based on a combination of cells and R-trees are: cell trees [46], P-trees [55], cell trees with oversize shelves [47] and general grid files [10].

In Table 2.2 we summarize the search costs of some of the data structures described, as well as the cost of storage and the cost of initial construction of the data structures from a file of n records.

Part III

DESIGN OF MULTIDIMENSIONAL DATA STRUCTURES

3. RANDOMIZED RELAXED K-D TREES

The K-d tree is a robust data structure that can handle a wide variety of queries. Its performance can be improved by means of static optimization techniques but this process requires the knowledge a priori of all the records in the file. So, optimized K-d trees are either useless or expensive in applications where files are required to be dynamic.

Another drawback of K-d trees is that in the average case analysis of their algorithms it is assumed that file records are uniformly distributed and that attributes are independent from each other, which is not always true in applications. Moreover, for trees built from files having the above characteristics a sequence of mixed-up insertions and deletions can distort the random properties of K-d trees.

A successful approach to overcome these problems consists in the use of randomization to produce simple and robust algorithms. In addition, randomization guarantees efficient expected performance that no longer depends on assumptions about the input distribution [75]. Randomization has been successfully applied to the design of dictionaries by Aragon and Seidel [1], Pugh [87] and Martínez and Roura [73]. A generalization of the approach of Aragon and Seidel to the case of multidimensional data structures is given by Mulmuley [76, 77], where randomization is applied to data structures in the field of computational geometry (Voronoi diagrams, for instance). However, in Mulmuley's work there is no support neither for dictionary operations nor for associative queries.

In this chapter we draw upon the ideas of Martínez and Roura to design randomized K-d trees, in order to present update algorithms which preserve the randomness of the tree, independently of the order of previous insertions and deletions. The main idea is to allow insertions and deletions in regions of K-d trees other than the leaves in such a way that the resulting tree is a random tree. As a consequence, the expected case performance of every operation holds irrespective of the order of update operations since it only depends on the random choices made by the randomized algorithms. A generalization of the approach of Aragon and Seidel to the case of K-d trees is also possible, yielding similar algorithms to the ones presented in this work, and with similar performances.

Randomized K-dimensional binary search trees (*randomized K-d trees*, for short), are thus a new type of K-dimensional binary trees that (1) support any sequence of update operations (insertions and deletions) while preserving the randomness of the tree, (2) do not demand preprocessing and (3) efficiently support exact match and associative queries. The results here

described are introduced in the paper *Randomized K-dimensional Binary Search Trees* [27].

3.1 Relaxed K-d Trees

Let us begin this section recalling that during the construction of a K-d tree, discriminants are assigned cyclically to each node (Condition 3 in Definition 2.2.1). Observe that this cyclic restriction of K-d trees forces update operations to be very laborious or located at the leaves of the tree. But, as we shall show, in our quest for randomized K-d trees, we need the flexibility of performing operations in places of K-d trees other than the leaves without major reorganization.

With this purpose in mind, we now introduce relaxed K-d trees. These are K-d trees where Condition 3 in Definition 2.2.1 is dropped but where each node explicitly stores its discriminant. So, the sequence of discriminants in a path from the root to any leaf is arbitrary.

Definition 3.1.1. A relaxed K-d tree for a set of K-dimensional keys is a binary tree in which:

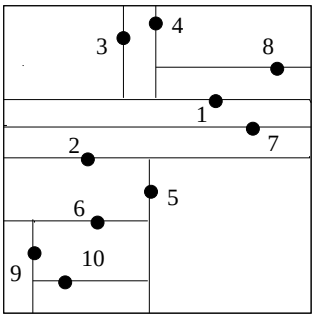
1. Each node contains a K-dimensional record and has associated a discriminant $j \in \{0, 1, \ldots, K-1\}$.

2. For every node with key x and discriminant j, the following invariant is true: any record in the right subtree with key y satisfies $y_j < x_j$ and any record in the left subtree with key y satisfies $y_j \geq x_j$.

Notice that if $K = 1$ a relaxed K-d tree is a binary search tree. Figure 3.1 shows a relaxed 2-d tree that results from the insertion of ten 2-dimensional keys into an initially empty tree. Each node contains its corresponding (key,discriminant) pair. The figure also shows the partition of $[0, 1]^2$ induced by the relaxed 2-d tree. It should be clear that if different discriminants had been chosen or if the same keys were inserted in different order the relaxed 2-d tree could be substantially different.

We say that a relaxed K-d tree of size n is *random* if it is built by n insertions where the keys are independently drawn from a continuous distribution (for example, uniformly from $[0, 1]^K$) and the discriminants are uniformly and independently drawn from $\{0, \ldots, K-1\}$.

$x_1 = (0.692, 0.703)$
$x_2 = (0.286, 0.495)$
$x_3 = (0.410, 0.895)$
$x_4 = (0.522, 0.953)$
$x_5 = (0.507, 0.394)$
$x_6 = (0.295, 0.300)$
$x_7 = (0.811, 0.605)$
$x_8 = (0.912, 0.807)$
$x_9 = (0.093, 0.210)$
$x_{10} = (0.188, 0.109)$



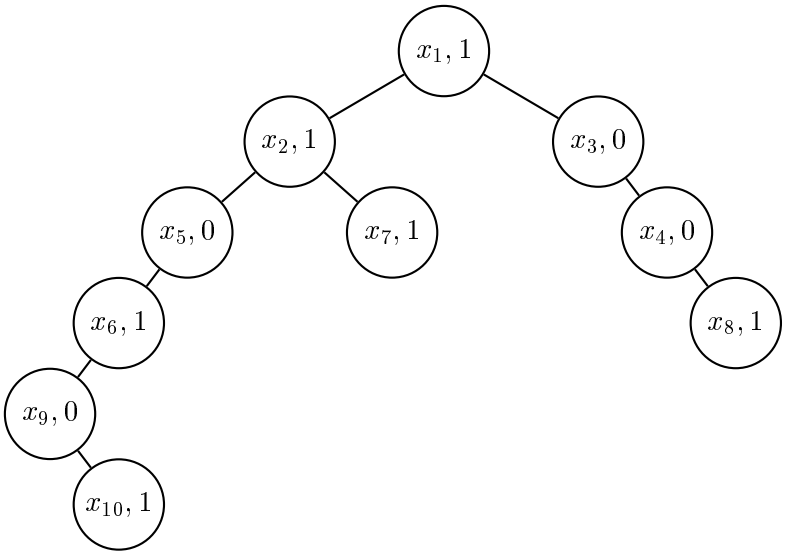


Fig. 3.1: A relaxed 2-d tree and the corresponding induced partition of $[0, 1]^2$.

In random relaxed K-d trees, this assumption about the distribution of the input implies that the $n!^K K^n$ possible configurations of input file and discriminant sequence are equally likely (observe that in random K-d trees the assumption implies that all the $n!^K$ input sequences are equally likely) [4, 38, 70].

In particular, in a random relaxed K-d tree each of the nK possibilities of (key, discriminant) pairs are equally likely to appear in the root and once the root is fixed, the left and right subtrees are independent random relaxed K-d trees. Thus, we obtain the same distribution for the shapes of random relaxed K-d trees as for the shapes of random binary search trees and the shapes of random K-d trees. Moreover, the distribution of random relaxed K-d trees of size n can be nicely characterized in terms of itself, and the probability that the left subtree of a random relaxed K-d-tree of size n has size l is $1/n$, for $0 \leq l < n$, independently of the dimension K.

Definition 3.1.2. T is a random relaxed K-d tree if and only if

1. $|\,T\,| = 0$ (i.e. T is empty) or,

2. $|\,T\,| = n$, its left and right subtrees are independent random relaxed K-d trees, and for any $x \in T$ and any discriminant j, $0 \leq j \leq K-1$,

$$\mathbb{P}\,[(x,j) \text{ is the root of } T] = \frac{1}{Kn}$$

Random K-d trees satisfy a similar (at least in spirit) but more complex recursive characterization, as the discriminant in successive levels must follow the cyclic sequence: $0, 1, \ldots, K-1, 0, 1, \ldots$

As we shall see, in Part IV, the recursive definition above for random relaxed K-d trees makes the analysis of partial match and other associative queries in relaxed K-d trees much easier than in standard K-d trees.

Parameters like the internal or the external path length (see [60, 70] for definitions) do not depend on the discriminants and hence their expected value is the same for random binary search trees and random relaxed K-d trees.

Theorem 3.1.1. ([60]) *The expected internal and external path length of a random relaxed K-d tree is $\Theta(n \log n)$. More precisely, the average internal path length I_n of a random relaxed K-d tree of size n is $I_n = 2(n+1)H_n - 4n$.*

Therefore, random relaxed K-d trees perform like random K-d trees for exact match queries and insertions. These operations explore a path that, because of Theorem 3.1.1, is of logarithmic length on the average.

The algorithms for associative queries in relaxed K-d trees are essentially the same as for K-d trees but their expected-case performances are not necessarily the same. For instance, partial match queries have not the same average complexity in K-d trees and in relaxed K-d trees (see Chapter 7) since for random relaxed K-d trees a path down the tree does not have a known cyclic pattern of discriminants but a random sequence of discriminants.

3.2 Randomized K-d Trees

Relaxed K-d trees have been shown to efficiently support dictionary operations and associative queries in the expected-case (see Chapters 7 and 8). However, this expected performance is based under the assumption that the input sequences of keys from which the tree is built correspond to a random permutation (that is, the $n!^K K^n$ possible configurations of input sequences and discriminants are equally likely). In what follows, using randomized algorithms, we will guarantee the expected performance, irrespective of this assumption about the input sequence. The randomized algorithms presented here require that each node stores the size of the subtree beneath it [73], because their behavior depends on the size of the subtrees to which they are applied.

We say that a relaxed K-d tree is a *randomized K-d tree* if it is the result of a sequence of update operations performed by means of the randomized algorithms introduced below, applied to an initially empty tree. We shall show in this section that any tree obtained this way is a random relaxed K-d tree.

Informally, in order to produce a random relaxed K-d tree two properties must hold (because of Definition 3.1.2):

1. A new inserted key should have the same probability of becoming the root of the tree, or the root of one of the subtrees of the root, and so forth and,

2. Every discriminant should have the same probability of being the discriminant of the new node.

Without loss of generality, we assume in what follows that randomized algorithms have free access to a source of random bits. This assumption is required by Algorithms 3 (insertion) and 7 (join) because at each recursive step these algorithms generate a random number in order to decide whether an insertion at the root should take place (insertion) or which one of two possible nodes should become the new root (join). We assume also that the cost of generating a random number of $\Theta(\log n)$ bits is constant [75]. We use this assumption for the analysis of the randomized operations, where we take as dominant cost the number of nodes visited by the randomized algorithms. Thus, the constant cost of generating a random number does not modify the algorithm's order of growth.

The randomized insertion of x in a relaxed K-d tree T, begins by generating uniformly a random integer, say j, from the set $\{0, \ldots, K-1\}$. This step corresponds to the assignment of a discriminant to the new node. If the tree T is the empty tree, then the algorithm insert produces a tree with root node (x, j) and empty left and right subtrees.

If the tree T is not empty, then, with probability $\frac{1}{n+1}$ the (key, discriminant) pair (x, j) must be placed at the root of the new tree using the insert_at_root algorithm (since the new tree will have size $n+1$). Otherwise, we insert the pair (x, j) recursively in either the left or the right subtree of T depending on x's order relation with the root of T. This procedure is depicted in Algorithm 3, where the function `random`$(0, n)$ returns an integer between 0 and n, both included.

Algorithm 3 The insertion algorithm for randomized relaxed K-d trees.

```
function insert(T : K-d tree, x : key) : K-d tree
    j := random(0, K − 1);
    if (T = nil) return new_node(x, j, nil, nil);
    i := T → discriminant;
    n := T → size;
    r := random(0, n);
    if (r = n) then
      return insert_at_root(T, x, j);
    if (x[i] < T → key[i]) then T → left := insert(T → left, x);
    else T → right := insert(T → right, x);
    return T;
end
```

The algorithm insert requires the possibility of inserting the pair (x, j)

at the root of any subtree of a relaxed K-d tree T, a task performed by Algorithm 4. If T is empty, then insert_at_root (T, x, j) gives as a result a tree with root node x, discriminant j and empty left and right subtrees. When T is not empty, by definition, insert_at_root $(T, x, j) = T'$, where the root of T' is (x, j), its left subtree consists of all those elements of T with j-th attribute smaller than x_j and its right subtree contains those elements of T with j-th attribute greater than x_j. To obtain the left and right subtrees of T' we use the split algorithm (which is described in Section 3.3).

Algorithm 4 The insertion at the root of randomized relaxed K-d trees.

```
function insert_at_root(T : K-d tree, x : key, j : integer) : K-d tree
    if (T = nil) then return new_node(x, j, nil, nil);
    S := split_<(T, x, j);
    G := split_>=(T, x, j);
    return new_node(x, j, S, G);
end
```

The deletion of a record from a relaxed K-d tree is presented in Algorithm 5. It consists of searching in the tree for the key x to be deleted (removing it if present) and then of joining its corresponding left and right subtrees (if required). In order to preserve a random relaxed K-d tree after deletion, all the nodes in the two subtrees of the deleted node should have some probability of taking the place of the deleted node. This is achieved by the join algorithm (Algorithm 7 below).

Algorithm 5 The deletion algorithm for randomized relaxed K-d trees.

```
function delete(T : K-d tree, x : key) : K-d tree
    if (T = nil) then return T;
    j := T → discriminant;
    if (x[j] < T → key[j]) then T → left := delete(T → left, x);
    else
      if (x > T → key[j]) then T → right := delete(T → right, x);
      else T := delete(join(T → left, T → right, j), x);
    return T;
end
```

Observe that both, insertions and deletions, consist of two different steps. A first step in which one must follow a path in the tree in order to locate the

place where the key must be inserted or deleted and, a second step in which the update is performed with either the insert_at_root or the join algorithms.

3.3 The Split and Join Algorithms

The insertion of a pair (x, j) at the root of a relaxed K-d tree T (the task performed by $\mathsf{insert\_at\_root}\,(T, x, j)$) is accomplished in two phases. First the tree T is partitioned with respect to the j-th attribute of x to produce two trees, $T_{<_j}$ and $T_{\geq_j}$, that contain all those keys of T whose j-th attribute is smaller (respectively greater or equal) than x_j. Second, the two trees of the previous step are attached to the root (x, j). But the core of $\mathsf{insert\_at\_root}\,(T, x, j)$ is clearly the partitioning or splitting process $\mathsf{split}\,(T, x, j) = \left(T_{<_j}, T_{\geq_j}\right)$. To simplify the description of split we will see it as a pair of functions $\mathsf{split}_<$ and $\mathsf{split}_\geq$ with $\mathsf{split}_<\,(T, x, j) = T_{<_j}$ and $\mathsf{split}_\geq\,(T, x, j) = T_{\geq_j}$. In practice, both $T_{<_j}$ and $T_{\geq_j}$ can be simultaneously computed.

The algorithm $\mathsf{split}_<$ works in the following way. When T is empty, the $\mathsf{split}_<$ algorithm returns an empty tree. Otherwise, let T have root (y, i), left subtree L, and right subtree R. We have four cases to consider:

1. If $j = i$ and $y_j < x_j$, then y belongs to $T_{<_j}$, all the elements of L do as well (since each j-th attribute in L is smaller than y_j and thus smaller than x_j) and the operation proceeds recursively in R to complete the result;

2. If $j = i$ and $y_j \geq x_j$, then the algorithm proceeds recursively in L (since y and all the elements in R are greater than x_j);

3. If $j \neq i$ and $y_j < x_j$, then y belongs to $T_{<_j}$ and the algorithm proceeds recursively in L and in R and attaches to y as left and right subtrees the results of splitting L and R respectively;

4. If $j \neq i$ and $y_j \geq x_j$, then y is not in the tree and the algorithm must proceed recursively in L and R, but now it has to join the trees resulting from splitting L and R.

The $\mathsf{split}_\geq$ operation is defined symmetrically. It is not difficult to see that both $\mathsf{split}_<\,(T, x, j)$ and $\mathsf{split}_\geq\,(T, x, j)$ compare x against at least the same keys in T as if we were performing a partial match with one attribute –the j-th attribute–specified. But the additional cost of the join algorithm must be taken into account. This fact is illustrated by next lemma.

Algorithm 6 The split$_<$ algorithm for randomized relaxed K-d trees.

```
function split_<(T : K-d tree, x : key, j : integer) : K-d tree
    if (T = nil) then return T;
    y := T → key;
    i := T → discriminant;
    if (j = i) then
      if (y[j] < x[j]) then T → right := split_<(T → right, x, j);
      else T := split_<(T → left, x, j);
    else
      if (y[j] < x[j]) then
        T → left := split_<(T → left, x, j);
        T → right := split_<(T → right, x, j);
      else T := join(split_<(T → left, x, j), split_<(T → right, x, j), j);
    return T;
end
```

Lemma 3.3.1. *Let T be a K-d tree, x a K-dimensional record and, w a bit-string in $\{0,1\}^K$ such that $w_j = 1$ and the rest of bits are zero. If y is a record in T visited by a partial match query with associated hyper-plane $H(x,w)$ then y is visited by* split*(T,x,j).*

Proof. Let T be a K-d tree. Let us consider the partial match algorithm with associated hyper-plane $H(x,w)$ on the tree T, where x is a K-dimensional key and w is a bit string in $\{0,1\}^K$ such that $w_j = 1$ and the rest of bits are zero. Thus, the partial match query has one coordinate specified. At any node y in T with discriminant i, the partial match algorithm proceeds as follows. At the first step it checks wether y satisfies the query (it visits y), exactly as do the split algorithm in order to decide wether y belongs to split$_<$ or to split$_\geq$. Then, if $i = j$, it proceeds in either the left or the right subtree of y if $x_j < y_j$ or $x_j \geq y_j$ respectively. The split algorithm proceeds splitting the same subtree in which the partial match proceeds the search. In contrast, if $i \neq j$, the partial match algorithm proceeds in both subtrees. The split algorithm does as well and additionally it joins two of the resultant trees. As a consequence, the split algorithm visits at least the same nodes that visits the partial match algorithm and the lemma follows. □

Lemma 3.3.1 provides a lower bound for the cost of the split algorithm. In general, this algorithm is applied to subtrees of average size $2 \log n$ [71].

We now describe the algorithm join whose input is a pair of relaxed K-d trees A and B and a discriminant j. By definition this algorithm is applied only when the j-th attribute values of the tree A are smaller than the j-th attribute values of the tree B. As we have already pointed out, in order to produce a random relaxed K-d tree, each node of A and each node of B must have some probability of becoming the root of the new tree $T = \mathsf{join}\,(A, B, j)$ (because of Definition 3.1.2).

If A and B have sizes n and m respectively, then, $T = \mathsf{join}\,(A, B, j)$ has size $n + m$. Hence, if $n > 0$ and $m > 0$ the join algorithm selects with probability $\frac{n}{n+m}$ the root of A, (a, j_a), as root of the resultant tree T and with complementary probability the root (b, j_b) of B. We have the following three cases to consider.

1. If A and B are both empty, then, T is the empty tree;

2. If only one of them is empty, then T is equal to the non-empty one (either A or B);

3. Let A and B be both non-empty trees with roots (a, j_a), (b, j_b), left subtrees L_a, L_b and right subtrees R_a and R_b, respectively. In this situation, if (a, j_a) is selected to become the root of T, then there are two subcases to consider:

   (a) If $j = j_a$, then the left subtree of T is L_a and its right subtree is the result of joining R_a with B (since by definition all the keys in B have j-th attributes greater than $a_{j_a} = a_j$);

   (b) If $j \neq j_a$, then the left subtree of T is the result of joining L_a with $\mathsf{split}_{<}\,(B, a, j_a)$ and the right subtree of T is the result of joining R_a with $\mathsf{split}_{\geq}\,(B, a, j_a)$.

Two analogous (and symmetrical) subcases arise if the root (b, j_b) of B is selected to become the root of T.

Observe that the join algorithm traverses the tree in a similar way than the partial match algorithm with one specified attribute, with the additional cost of the split algorithm. Observe also, that this algorithm is generally applied to subtrees of expected size $2 \log n$ [71].

Algorithm 7 The join algorithm for randomized relaxed K-d trees.

```
function join(A : K-d tree, B : K-d tree, j : integer) : K-d tree
    if (A = nil) then return B;
    if (B = nil) then return A;
    a := A → key; j_a := A → discriminant; L_a := A → left; R_a := A → right;
    b := B → key; j_b := B → discriminant; L_b := B → left; R_b := B → right;
    m := A → size; n := B → size; total := n + m;
    r := random(0, total − 1);
    if (r < m) then
      T := A;
      if (j = j_a) then T → right := join(R_a, B, j);
      else
        T → left := join(L_a, split_<(B, a, j_a), j);
        T → right := join(R_a, split_>=(B, a, j_a), j);
    else
      T := B;
      if (i = j_b) then T → left := join(A, L_b, j);
      else
        T → left := join(split_<(A, b, j_b), L_b, j);
        T → right := join(split_>=(A, b, j_b), R_b, j);
    return T;
end
```

3.4 Properties of Randomized K-d Trees

The randomized split and join algorithms preserve the randomness of their input (see Lemma 3.4.1). In other words, when applied to random relaxed K-d trees, both the split and the join algorithms produce random relaxed K-d trees. Moreover, since this happens, the insert and delete algorithms when applied to randomly built relaxed K-d trees produce random relaxed K-d trees. All these claims are made explicit in Lemma 3.4.1, Theorem 3.4.2 and Theorem 3.4.3. We recall that the definition of random K-d trees implies that the set of keys stored in the tree are all compatible.

Lemma 3.4.1. *Let T be a relaxed K-d tree and let $T_{<_j}$ and $T_{\geq_j}$ be the relaxed K-d trees produced by* $\mathsf{split}_< (T, x, j)$ *and* $\mathsf{split}_\geq (T, x, j)$*, respectively, where x is any key compatible with T. Then, if T is a random relaxed K-d tree, $T_{<_j}$ and $T_{\geq_j}$ are independent random relaxed K-d trees.*

Let T' be the relaxed K-d tree produced by $\mathsf{join}\,(A, B, j)$*, where A and B are relaxed K-d trees such that, for all keys x of A and all keys y of B,*

$x_j < y_j$. Then, if A and B are independent random relaxed K-d trees, T' is a random relaxed K-d tree.

Proof. We prove the two parts of this lemma by induction: on the size n of T to show that the algorithms $\mathsf{split}_<$ and $\mathsf{split}_\geq$ preserve randomness, and on the joint size $n = |A| + |B|$ of T' to show that algorithm join also preserves randomness. Observe that to prove the two parts of the lemma for the size n, we will need to inductively and simultaneously assume that both statements are true if T (T') is of size smaller than n. The reason for this is the mutual recursion between the split and the join operations.
If T is empty ($n = 0$) then both $\mathsf{split}_< (T, x, j)$ and $\mathsf{split}_\geq (T, x, j)$ are empty trees. Also, if A and B are empty ($n = |A| + |B| = 0$) then T' is empty and hence random. And hence the first part of the lemma trivially holds for the basis of the induction.
Let us consider now the case where $n > 0$, assuming now that both parts of the lemma are true for all sizes smaller than n.
Let us consider first the splitting process. Let n denote the size of T, the relaxed K-d tree to be partitioned. Let the root of T be (y, i) and L, R denote its left and right subtrees, respectively.
If $j = i$ and $y_j < x_j$, then the root (y, i) belong to the tree $T_{<_j}$, the tree $T_{\geq_j}$ and the right subtree R' of the tree $T_{<_j}$ are computed by applying recursively $\mathsf{split}_<$ and $\mathsf{split}_\geq$ to subtree R. Since $|R| < n$, by the inductive hypothesis, both $T_{\geq_j}$ and R' are random and independent. The left subtree of $T_{<_j}$ is L, the left subtree of T. Since T is assumed to be random, L must be, by definition, random and independent of R. Therefore L is also independent of R'. In summary, both subtrees of $T_{<_j}$ are random and independent relaxed K-d trees, and $T_{<_j}$ and $T_{\geq_j}$ are independent.
To complete the proof for this case, we need only to show that, for every key z in $T_{<_j}$ and every discriminant $j' \in \{0, 1, \ldots, K-1\}$, the probability that the pair (z, j') is at the root of $T_{<_j}$ is $1/Km$ where m is the size of $T_{<_j}$. Indeed,

$$\begin{aligned}
&\mathbb{P}\left[\{(z, j') \text{ is root of } T_{<_j} \mid y_j < x_j \text{ and } j = i\}\right] \\
&\quad = \frac{\mathbb{P}\left[\{(z,j') \text{ is root of } T \text{ and } y_j<x_j \text{ and } j=i\}\right]}{\mathbb{P}\left[\{y_j<x_j \text{ and } j=i\}\right]} \\
&\quad = \frac{\frac{1}{Kn}\frac{1}{K}}{\frac{m}{n}\frac{1}{K}} = \frac{1}{Km}
\end{aligned}$$

The case in which $j = i$ and $y_j \geq x_j$ can be proved in a similar way. Consider now the case in which $j \neq i$ and $y_j < x_j$. In this case the split process is

recursively applied to both L and R, yielding by inductive hypothesis four independent random relaxed K-d trees: $L_{<_j}$,$R_{<_j}$, $L_{\geq_j}$ and $R_{\geq_j}$. The tree $T_{<_j}$ is built by attaching the trees $L_{<_j}$ and $R_{<_j}$ to the root (y, i) and the tree $T_{\geq_j}$ is obtained by joining the trees $L_{\geq_j}$ and $R_{\geq_j}$ with respect to i, because all the nodes of L have i-th coordinates less than the i-th coordinates of the nodes of R. As $|\, L_{\geq_j} \,| + |\, R_{\geq_j} \,| < n$, we have that $T_{\geq_j} = \mathsf{join}\left(L_{\geq_j}, R_{\geq_j}, i\right)$ is, by inductive hypothesis, a random relaxed K-d tree, clearly independent of $T_{<_j}$. Furthermore, the probability that for any key z in $T_{<_j}$ and any discriminant $j' \in \{0, 1, \ldots, K-1\}$, the pair (z, j') is the root of $T_{<_j}$ given that $j \neq i$, is $1/Km$ (where m is the size of $T_{<_j}$), this is,

$$\begin{aligned}
&\mathbb{P}\left[\{(z, j') \text{ is root of } T_{<_j} \mid y_j < x_j \text{ and } j \neq i\}\right] \\
&\quad = \frac{\mathbb{P}\left[\{(z,j') \text{ is root of } T \text{ and } y_j < x_j \text{ and } j \neq i\}\right]}{\mathbb{P}\left[\{y_j < x_j \text{ and } j \neq i\}\right]} \\
&\quad = \frac{\frac{1}{Kn}\left(1 - \frac{1}{K}\right)}{\frac{m}{n}\left(1 - \frac{1}{K}\right)} \\
&\quad = \frac{1}{Km}
\end{aligned}$$

Now we tackle the second part of the lemma and show that join preserves randomness when $n > 0$. If either A or B are empty, then join returns the non-empty tree in the pair which, by hypothesis, is random. Thus we shall only consider the case where both $|\, A \,| > 0$ and $|\, B \,| > 0$. Let A have root (a, j_a), left subtree L_a and right subtree R_a, and let B have root (b, j_b), left subtree L_b and right subtree R_b. If we select the pair (a, j_a) to become the root of T', and $j = j_a$ then we will recursively join R_a and B. By the inductive hypothesis, the result is a random relaxed K-d tree. Thus, we have that the left subtree of T' is L_a, which is a random relaxed K-d tree and that the right subtree is the tree returned by $\mathsf{join}\,(R_a, B, j)$, which is also random. Both subtrees are independent because A and B are independent and the probability that (a, j_a) was the root of L_a times the probability that it is selected as root of T' is $\frac{1}{K|\, A \,|}\frac{|\, A \,|}{|\, A \,|+|\, B \,|} = \frac{1}{Kn}$. The same reasoning shows that the lemma is true when (b, j_b) is chosen as the root of T' and $j = j_b$.
Finally, if (a, j_a) is chosen as the root of T' but $i \neq j_a$ then, we need first to split B with respect to a and j_a, yielding by the inductive hypothesis two independent random relaxed K-d trees: $B_{<_{j_a}} = \mathsf{split}_<\,(B, a, j_a)$ and $B_{\geq_{j_a}} = \mathsf{split}_\geq\,(B, a, j_a)$. Then, join is recursively applied to the pairs $\left(L_a, B_{<_{j_a}}\right)$ and $\left(R_a, R_{\geq_{j_a}}\right)$ yielding by inductive hypothesis two random independent relaxed K-d trees that are attached to the root (a, j_a) as left and right subtrees respectively. The probability that (a, j_a) is the root of T' is $\frac{1}{K(|\, A \,|+|\, B \,|)} = \frac{1}{Kn}$

(for the same reason that the case in which (a, j_a) was selected as the root but $j = j_a$). We have a similar behavior if (b, j_b) were chosen as the root of T' and $j \neq j_b$, then the lemma holds, by reasoning as before. □

Theorem 3.4.2. *If T is a random relaxed K-d tree that contains the set of keys X and x is any key compatible with X, then* insert(T, x) *returns the random relaxed K-d tree containing the set of keys $X \cup \{x\}$.*

Proof. The proof is by induction on the size n of T. If $n = 0$, then T is the empty tree (a random relaxed K-d tree), and insert(T, x) returns a random relaxed K-d tree with root (x, j), and two empty subtrees, where j is uniformly generated from $\{0, \ldots, K-1\}$. We assume now that T is not empty and that the theorem is true for all sizes $< n$. The insertion of (x, j) in T have two possible results, with probability $\frac{1}{K(n+1)}$, (x, j) is the root of $T' =$ insert (T, x) and with complementary probability (x, j) is inserted in the corresponding left or right subtree of T.
Let us consider first the case in which (x, j) is not inserted at the root of T. Consider an item (y, i) already in T. The probability that (y, i) is the root of T before the insertion of x is $\frac{1}{Kn}$ since, by hypothesis, T is a random relaxed K-d tree. The probability that (y, i) is at the root of T' is the probability that x is not at the root of T' and (y, i) was at the root of T, which is $\frac{1}{Kn} \times \frac{n}{n+1}$, resulting in the desired probability. Moreover, since (x, j) is not inserted at the root of T' at the first step, the insertion proceeds recursively in either the left or the right subtrees of T, which are independent random relaxed K-d trees of sizes $< n$. One of them is not modified during the insertion and is a random relaxed K-d tree, and by the inductive hypothesis, the other one, after insertion is also a random relaxed K-d tree. Thus, T' is a random relaxed K-d tree of size $n + 1$.
On the other hand, with probability $\frac{1}{K(n+1)}$, (x, j) becomes the root of T'. The tree T, which by hypothesis is a random relaxed K-d tree, must be split with respect to (x, j), and because of Lemma 3.4.1 this step produces two independent random relaxed K-d trees, which are the left and right subtrees of T', thus T' is also a random relaxed K-d tree. □

Theorem 3.4.3. *If T is a random relaxed K-d tree that contains the set of keys X, then,* delete (T, x) *produces a random relaxed K-d tree T' that contains the set of keys $X \backslash \{x\}$.*

Proof. If the key x is not in T, then the algorithm does not modify the tree, which by hypothesis is a random relaxed K-d tree. Let us now suppose that x is in T, this case is proved by induction on the size of the tree. If $n = 1$, x is the only key in the tree and after deletion we obtain the empty tree which is a random relaxed K-d tree. We now assume that $n > 1$ and that the theorem is true for all sizes smaller than n. If x was not the key at the root of T we proceed recursively either in the left or in the right subtrees, and by inductive hypothesis we obtain a random subtree. If x was the key at the root of T, the tree after deletion is the result of joining the left and right subtrees of T, which produce a random relaxed K-d tree because of Lemma 3.4.1. Finally, after deletion, each node has probability $\frac{1}{K(n-1)}$ of being the root. Indeed, let (y, j) be any (key,discriminant) pair in T such that $y \neq x$,

$$\begin{aligned}
&\mathbb{P}\left[\{(y,j) \text{ is the root of } T'\}\right] \\
&\quad = \mathbb{P}\left[\{(y,j) \text{ is the root of } T' \mid x \text{ was not the root of } T\}\right] \\
&\qquad \times \mathbb{P}\left[\{x \text{ was not the root of } T\}\right] \\
&\qquad + \mathbb{P}\left[\{(y,j) \text{ is the root of } T' \mid x \text{ was the root of } T\}\right] \\
&\qquad \times \mathbb{P}\left[\{x \text{ was the root of } T\}\right] \\
&\quad = \tfrac{1}{K(n-1)}\tfrac{n-1}{n} + \tfrac{1}{K(n-1)}\tfrac{1}{n} \\
&\quad = \tfrac{1}{K(n-1)}
\end{aligned}$$

□

Combining the two previous theorems we obtain the following important corollary.

Corollary 3.4.4. *The result of any arbitrary sequence of insertions and deletions, starting from an initially empty tree is always a random relaxed K-d tree.*

It is important to observe that using these randomized update operations we obtain relaxed K-d trees for which the expected running time for insertions, deletions, partial match, orthogonal range, and nearest neighbor queries depend only on the random choices made by the randomized update algorithms and not on the sequence of updates.

4. RANDOMIZED QUAD TREES

Quad trees are frequently used in applications because they support a large set of operations with simple algorithms (except for deletions) and reasonable time requirements. As long as the dimension K is not too big (smaller than 4), the space requirements are also acceptable.

But unfortunately, quad trees suffer also from the same drawbacks mentioned for K-d trees. Their average-case analysis is made under probabilistic assumptions of independence and uniformity of keys. Sequences of insertions and deletions can distort random quad trees [21, 32]. The maintenance of dynamic files is expensive.

In this chapter we apply the ideas of [27, 73] to randomize quad trees, in order to present (as we did for K-d trees in Chapter 3) update algorithms that preserve the randomness of the tree, independently of the order of previous insertions and deletions. The resulting update algorithms are simple and defined for any dimension K. The case of deletions is of particular interest because the classic deletion algorithm over quad trees [93] is rather complex and its description becomes more complicated as the dimension increases.

This chapter is organized as follows. In Section 4.1 we give the required preliminaries of quad trees, with special emphasis in the deletion algorithm. In Section 4.2 we introduce randomized insertion and deletion algorithms and we give their properties in Section 4.3, where we prove that the randomized algorithms presented here always produce random quad trees. All the results described in this chapter are included in the article *Randomized Insertion and Deletion in Point Quad Trees* [26].

4.1 Quad Trees

In order to be able to give a definition of the randomized update algorithms for K-dimensional quad trees, we introduce the relation $\prec_w$ between K-dimensional keys.

Given a bit string $w = w_0 w_1 \cdots w_{K-1}$ of length K (that is, $w \in \{0,1\}^K$) and two K-dimensional keys x and y we say that x is related to y under the relation $\prec_w$, and write $x \prec_w y$, if and only if, for all $i \in \{0, 1, \ldots, K-1\}$, $x_i < y_i$ whenever $w_i = 0$ and $x_i \geq y_i$ whenever $w_i = 1$. Otherwise we say that $x \not\prec_w y$. Note that the relation $\prec_w$ is antisymmetric and transitive. Note also that if the keys x and y are compatible, then for every $w \in \{0,1\}^K$, if $x \prec_w y$, then $y \prec_{\overline{w}} x$, where $\overline{w}$ is the complementary bit string of w in $\{0,1\}^K$, that is, $\overline{w}_i = 0$ if and only if $w_i = 1$ ($i \in \{0, 1, \ldots, K-1\}$). Furthermore, $x \prec_w y$

for exactly one bit string $w \in \{0,1\}^K$ and $x \not\prec_w y$ for all other bit strings. We are now ready to give the definition of K-dimensional quad trees that we will use in what follows. It is a straightforward generalization of Definition 2.3.1 of two-dimensional quad trees.

Definition 4.1.1. A quad tree T for a set of K-dimensional keys is a tree in which:

1. Each node contains a K-dimensional key and pointers to 2^K subtrees, namely t_w for all $w \in \{0,1\}^K$.

2. For every node with key y the following invariant is true: any record with key x in the w-th subtree of y satisfies $x \prec_w y$.

Abusing language, we will use $w(i)$ to denote the bit string corresponding to the binary representation of the non-negative integer i. We will also occasionally say w-th subtree or w-th quadrant referring to subtree w or hyper-quadrant w irrespectively.

The standard insertion algorithm immediately follows from the previous definition: compare the new elements' key with the key stored at the root of the quad tree and obtain the index w of the subtree where the insertion must recursively proceed.

The deletion algorithm, first introduced by Samet [93], is not so straightforward. Analogous to deletion in binary search trees, the goal is to replace the deleted node with the closest possible one. But in quad trees it is not clear which of the remaining nodes is the closest in all dimensions simultaneously, and no matter which one is chosen, a reorganization of some nodes will be required. Here is the description of the algorithm for the two-dimensional case (it becomes more complex as the dimension increases): let x be the node to be deleted and y the node that is going to replace it. Ideally we would like to choose y such that the region R (in grey in Figure 4.1(a)) between the orthogonal lines passing through x and y is empty. Unfortunately, this is not always possible. In a two-dimensional quad tree there are four nodes (one for each subtree of x) that are candidates to be closest to x (see [93]). So, the first step of the algorithm is to find them. For each subtree t_w of x, the candidate node corresponding to t_w is found starting the search at the root node of t_w and systematically following the $\overline{w}$-th subtree until a node with empty $\overline{w}$-th subtree is reached (the node that we are looking for). Such a node is shown in Figure 4.1(b). The second step of the algorithm consists of

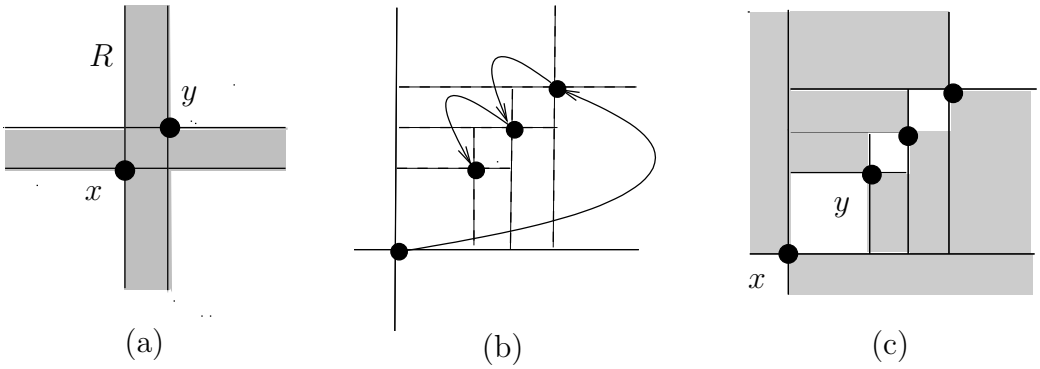


Fig. 4.1: Some aspects of Samet's deletion algorithm: (a) the desired empty area R (in grey), (b) the traversal of one quadrant for finding a replacement node, (c) the processed sub-quadrants by the adj_quad procedure (in grey), when node x is deleted and replaced by node y.

choosing from these four candidates the node y that will replace the deleted node x. This is done applying the following criteria.

- **Criterion 1**. Choose the node that is closer to each of its bordering axes than any other candidate on the same side of these axes, if such a candidate exists.
- **Criterion 2**. If no candidate satisfies Criterion 1, then the candidate with the minimum L_1 metric value (sum of the displacements from the bordering axes) should be chosen.

Once the replacement node y is chosen, the next step of the algorithm is to determine which are the nodes of the tree that require reinsertion. This is done by means of two procedures: adj_quad (adjacent quadrant) and new_root.

Let t_w be the subtree of x (the node to be deleted) that contains y (the replacement node). Note that no node of subtree $t_{\overline{w}}$ needs reinsertion. But the two subtrees adjacent ($\overline{w}$ is the opposite) to subtree w must be processed separately by the adj_quad procedure (see Figure 4.1(c), where the grey subtrees depict the subtrees that must be processed by adj_quad). The adj_quad procedure applied to a tree t_w examines the root of the tree, say r. If r is outside the region R, then the whole subtree must be reinserted in the new quad tree. Otherwise, two of its subtrees can automatically remain in

the w-th subtree with no need of more processing while the two remaining subtrees must be separately processed by a recursive call of the adj_quad procedure. Once the nodes in subtrees adjacent to t_w have been processed, it is the turn of nodes in t_w. Clearly all the nodes in (the white regions in Figure 4.1(c)) sub-subtree w of t_w will retain their position. So, procedure new_root is applied to the remaining subtrees of t_w.

The new_root procedure begins by applying procedure adj_quad to the subtrees adjacent to t_w and continues by iteratively reapplying new_root to sub-subtrees $\overline{w}$ of subtree w until an empty subtree is found in this direction (let us observe that the node that satisfies this condition is the replacing node y). The nodes in the subtrees adjacent to subtree w of the subtree rooted at y must be reinserted in the subtrees adjacent to subtree t_w of x. Because of the properties of the selection of candidate nodes the subtree $\overline{w}$ of y is empty. Also, subtree w of y replaces subtree $\overline{w}$ of x.

Theoretical and empirical results for the above deletion algorithm [93] show that, comparing against the complete reconstruction of the subtree beneath the deleted node, for independently and uniformly distributed data in $[0,1]^K$, the average number of nodes requiring reinsertion is reduced by a factor of 5/6 (83%) when the replacement node satisfies Criteria 1 and 2. When the replacement node is randomly chosen among the four possible candidates the average number of nodes requiring reinsertion is reduced by a factor of 2/3 (67%). This implies that this deletion algorithm has the same order of growth as a complete reconstruction of the subtree whose root is to be deleted. The expected cost is thus $\Theta(\log n \cdot \log\log n)$ for a tree of size n, since the expected size of the subtree beneath the deleted node is $2\log n$ [71].

Empirical tests show that the number of comparisons required to locate the deleted node in a quad tree of n nodes is proportional to $\log n$ and that the total path length of the tree after a single deletion decreases slightly. However, as it is shown in [32], for long sequences of insertions and deletions the total path length and the deletion cost augment.

4.2 Randomized Quad Trees

The randomized insertion and deletion algorithms presented here require that each node stores the size of the subtree beneath it [73], because their behavior depends on the size of the (sub)trees to which they are applied.

We assume, as in previous chapter, that randomized algorithms have

free access to a source of random bits and that the cost of generating a random number of $\Theta(\log n)$ bits is constant [75]. During their execution, the Algorithms 8 (randomized insertion) and 12 (join), described below, generate a random number in order to decide the place of an insertion or to choose the replacement node of a deleted one. The fact of assuming that the generation of such random numbers is constant implies that for the average-case analysis of the cost of these algorithms it suffices to take into account the number of visited nodes, just the way as this kind of analysis is usually done.

In exactly the same vein of randomized K-d trees, we would like insertion and deletion algorithms that produce quad trees that behave as if they were built by successive insertions of uniformly and independently generated multidimensional keys.

In order to produce such quad trees it is required that any new inserted key has some probability of becoming the root of the tree, or the root of one of the subtrees of the root, and so forth. Similarly, when a node is deleted it is required that all the remaining nodes in the subtree beneath it have some probability of replacing it. The randomized insertion and deletion algorithms that we present now provide these capabilities.

The randomized insertion algorithm (described as Algorithm 8) of a key x in a quad tree T ($\mathsf{insert}(T, x)$), proceeds as follows.

1. If the tree T is empty, then the algorithm insert produces a tree with root node x and 2^K empty subtrees.

2. If the tree T is not empty and has size $n > 0$, then, with probability $\frac{1}{n+1}$ the key x must be placed at the root of the new tree using the insert_at_root algorithm (since the new tree will have size $n + 1$). Otherwise, we insert x recursively in the corresponding w-th subtree of T depending on x's order relation with the root of T.

The algorithm insert requires the possibility of inserting the key x at the root of any subtree of a quad tree T.

1. If T is empty, then, $\mathsf{insert\_at\_root}(T, x)$ gives as a result a tree with root node x, and empty subtrees.

2. When T is not empty, $\mathsf{insert\_at\_root}(T, x) = T'$ where, by definition, the root of T' is x and, its w-th subtree consists of all those elements

Algorithm 8 The insertion algorithm for randomized quad trees.

```
function insert(T : quad tree, x : key) : quad tree
    if (T = nil) then return new_node(x);
    n := T → size;
    r := random(0, n);
    if (r = n) then return insert_at_root(T, x);
    for (i = 0; i < K; i++)
        if (x[i] < T → key[i]) then w[i] := 0;
        else w[i] := 1;
    T → w := insert(T → w, x);
    return T;
end
```

z of T such that $z \prec_w x$. To obtain the subtrees of T' we use the split algorithm which we present in Algorithm 11.

Algorithm 9 The insertion at the root of randomized quad trees.

```
function insert_at_root(T : quad tree, x : key) : quad tree
    if (T = nil) then return new_node(x);
    U := new_node(x);
    for (w ∈ {0, 1}^K)
        U → w := split_w(T, x);
    return U;
end
```

The deletion of a record from a random quad tree involves searching the key x to be deleted in the tree, and then joining its corresponding 2^K subtrees. Since it is required that all the nodes in these subtrees have some probability of taking the place of the deleted node, we require the join algorithm (introduced in Algorithm 12) which achieves this capability.

Observe that both insertions and deletions consist of two different steps. A first step in which one must follow a path in the tree in order to locate the place where the key must be inserted or deleted and, a second step in which the update is performed.

The insertion of a key x at the root of a tree T (the task that performs insert_at_root(T, x)) is accomplished in two steps: a first step in which the tree T is partitioned with respect to x producing the 2^K trees T'_u, for all $u \in \{0,1\}^K$, where T'_u contains all those keys z of T such that $z \prec_u x$; and a

Algorithm 10 The deletion algorithm for randomized quad trees.

```
function delete(T : quad tree, x : key) : quad tree
    if (T = nil) then return T;
    if (x = T → key) then T := delete(join(T → w(0), ..., T → w(2^K − 1)), x);
    else
      for (w ∈ {0,1}^K)
          if (x ≺_w T → key) then T → w := delete(T → w, x);
    return T;
end
```

second step in which the 2^K subtrees are attached to the new root x. Clearly the main cost of $\mathsf{insert\_at\_root}(T, x)$ lies in the partitioning or splitting process $\mathsf{split}(T, x)$.

To simplify the description of the split algorithm we will see it as a 2^K-tuple of functions split_u, for each $u \in \{0,1\}^K$, with $T'_u = \mathsf{split}_u(T, x)$. In practice all the T'_u trees can be simultaneously computed.

The algorithm split_u works in the following way. When T is empty, the split_u algorithm returns the empty tree. Otherwise, let T have root y and subtrees T_w for all $w \in \{0,1\}^K$. For each $u \in \{0,1\}^K$ we have the following cases to consider.

1. If $y \prec_w x$ and $w = u$, then y belongs to T'_u, all the elements of T_w do as well (because $\prec_w$ is transitive), and the operation proceeds recursively in the rest of subtrees in order to complete the result;

2. If $y \prec_w x$ and $w \neq u$, then y does not belong to T'_u, neither do all the elements of T_w (because $\prec_w$ is transitive), and the operation proceeds recursively in the remaining subtrees producing $2^K - 1$ subtrees that must be combined by means of the `join` algorithm.

It is not difficult to see that $\mathsf{split}(T, x)$ compares x against the keys of $2^K - 1$ subtrees of T. However, this observation does not provide a characterization of the cost of split because the algorithm may visit some nodes more than once. Hence, this is only a lower bound for the number of visited nodes. The additional cost of the join algorithm must be also taken into account.

It is also worth to observe that, in the average, the algorithm split is applied to relatively small subtrees, that is of size $2 \log n$ in average [71].

Algorithm 11 The split_u algorithm for randomized quad trees.

```
function split_u(T : quad tree, x : key) : quad tree
    if (T = nil) then return T;
    y := T → key;
    for (w ∈ {0,1}^K)
        if (y ≺_w x) then
            if (w = u) then
                T'_u → key := y;
                T'_u → w := T_w;
                for (v ∈ {0,1}^K, v ≠ w)
                    T'_u → v := split_u(T → v, x);
            else T'_u := join(split_u(T → w(0), x), ..., split_u(T → w(2^K − 1), x));
    return T'_u;
end
```

We now describe the algorithm $\mathsf{join}(T_{w(0)}, T_{w(1)}, \ldots, T_{w(2^K-1)})$. By definition this algorithm is applied only when the keys in the trees $T_{w(i)}$ are related to a key y by relations $\prec_{w(i)}$ respectively (with $0 \leq i \leq 2^K - 1$).

As we have already pointed out, in order to produce random quad trees, each node of the trees $T_{w(0)}, T_{w(1)}, \ldots, T_{w(2^K-1)}$ should have some probability of becoming the root of the new tree.

Let the sizes of $T_{w(0)}$, $T_{w(1)}$,..., $T_{w(2^K-1)}$ be $n_0, n_1, \ldots, n_{2^K-1}$ respectively. If $T = \mathsf{join}(T_{w(0)}, T_{w(1)}, \ldots, T_{w(2^K-1)})$, then T has size $n = n_0+n_1+\ldots+n_{2^K-1}$. The join algorithm selects, with probability $\frac{n_i}{n}$, the root of $T_{w(i)}$ as root of T, and there are two cases to consider.

1. If the trees $T_{w(0)}$, $T_{w(1)}$,..., $T_{w(2^K-1)}$ are all empty, then the tree $T = \mathsf{join}(T_{w(0)}, T_{w(1)}, \ldots, T_{w(2^K-1)})$ is also empty;

2. If at least one of them is not empty and $T_{w(0)}, T_{w(1)}, \ldots, T_{w(2^K-1)}$ have roots $y_0, y_1, \ldots, y_{2^K-1}$ respectively with subtrees $t^0_{u(j)}, t^1_{u(j)}, \ldots, t^{2^K-1}_{u(j)}$ for $j = 0, \ldots, 2^K - 1$, respectively. Then, if y_i is selected as the root of the tree T we have the following.

   (a) All the keys of the trees $t^i_{u(j)}$ should be members of the $w(j)$-th subtree of T for all $j \in \{0, 1, \ldots, 2^K - 1\}$.

   (b) If the keys in T are compatible then, the whole tree $T_{\overline{w(i)}}$ becomes part of the $\overline{w(i)}$-th subtree of T, since, for every $z \in T_{\overline{w(i)}}$, $z \prec_{\overline{w(i)}} y \prec_{\overline{w(i)}} y_i$ (where y is the deleted node).

(c) The remaining trees must be split with respect to y_i and the corresponding trees joined together.

Algorithm 12 The join algorithm for randomized quad trees.

function `join`($T_{w(0)}$: quad tree, $T_{w(1)}$: quad tree, $\ldots$, $T_{w(2^K-1)}$: quad tree) : quad tree
  for $(i = 0, \ldots, 2^K - 1)$
    $y_i := T_{w(i)} \rightarrow key$;
    $n_i := T_{w(i)} \rightarrow size$;
    for $(j = 0, \ldots, 2^K - 1)$
      $t^i_{u(j)} := T_{w(i)} \rightarrow u(j)$;
  $total := n_0 + n_1 + \ldots + n_{2^K-1}$;
  $i := -1$;
  $sum := 0$;
  $r :=$ `random`$(0, total - 1)$;
  while $(r \geq sum)$
    $i++$;
    $sum+ = n_i$;
  $T \rightarrow key := y_i$;
  for $(w \in \{0,1\}^K)$
    $T \rightarrow w :=$ `join`(`split_w`$(T_{w(0)}, y_i), \ldots, t^i_{u(j)}, \ldots,$ `split_w`$(T_{w(2^K-1)}, y_i)$);
  return T;
end

Observe that, the join algorithm traverses the tree in a similar way than the split algorithm, since the join algorithm is continued recursively in $2^K - 1$ subtrees (in fact $2^K - 2$ if the keys are all compatible). This is a lower bound for the number of visited nodes of the algorithm. The additional cost of the split algorithm in $2^K - 1$ subtrees should also be taken into account. In average, the join algorithm is applied near the leaves, in subtrees of expected logarithmic size [71].

We say that a quad tree is a *randomized quad tree* if it is the result of a sequence of update operations performed by means of the randomized algorithms introduced below, applied to an initially empty tree. We shall show in the next section that any quad tree obtained this way is a random quad tree.

4.3 Properties of Randomized Quad Trees

In this section we prove that the result of any arbitrary sequence of randomized insertions and deletions starting from an initially empty quad tree is always a random quad tree. This result implies that the theoretical results given in the literature for random quad trees hold in practice when the randomized algorithms presented here are used.

A K-dimensional quad tree of size n is *random* if it is built by n insertions of K-dimensional keys independently drawn from a continuous distribution in $[0,1]^K$ (for simplicity let us assume uniform distribution). This assumption about the distribution of the input implies that the $n!^K$ distinct configurations of input sequences are equally likely. In particular, in a random quad tree of size n, each of the n possible K-dimensional keys are equally likely to appear in the root, and once the root is fixed, the 2^K subtrees are independent random quad trees.

Observe that, unlike binary search trees, the external nodes of a K-dimensional quad tree are not equally likely to be the position of the next insertion. However, observe that, for $n > 2$, in an input sequence of n K-dimensional keys, the last key can not be in the root of a quad tree, it must belong to one of the 2^K subtrees. Given n_j, the size of the j-th subtree after n insertions ($j = 0, 1, \ldots, 2^K - 1$), any of the n_j keys could be the last one and thus the last key is in the j-th subtree with probability $n_j/(n-1)$ [70].

The randomized split and join algorithms preserve the randomness of their input (Lemma 4.3.1 below). In other words, when applied to random quad trees, both the split and the join algorithms produce random quad trees. Moreover, since this happens, the insert and delete algorithms when applied to random quad trees produce random quad trees. These claims are formalized as follows.

Lemma 4.3.1. *i) Let T be a quad tree and let $T_{u(i)}$ (for all $u(i) \in \{0,1\}^K$) be the quad trees produced by* $\mathsf{split}(T, x)$*, where x is any key compatible with T. Then, if T is a random quad tree, the $T_{u(i)}$ trees are independent random quad trees.*
ii) Let T' be the quad tree produced by $\mathsf{join}(T_{w(0)}, T_{w(1)}, \ldots, T_{w(2^K-1)})$*, where $T_{w(i)}$ and $T_{\overline{w(i)}}$ are quad trees such that, for all keys x of $T_{w(i)}$ and all keys y of $T_{\overline{w(i)}}$, $x \prec_{w(i)} y$. If the $T_{w(i)}$ trees are independent random quad trees then T' is a random quad tree.*

Proof. We prove the two parts of this lemma by induction on the size n of T

to show that split preserves randomness, and on the joint size $n = |T_{w(0)}| + |T_{w(1)}| + \ldots + |T_{w(2^K-1)}|$ of T' to show that join also preserves randomness. Observe that to prove the two parts of the lemma for size n, we will need to inductively and simultaneously assume that both statements are true if T (T') is of size smaller than n. The reason is the mutual recursion between the split_u operations and join.

If T is empty ($n = 0$) then $\mathsf{split}_u(T, x)$ is an empty tree for all $u \in \{0,1\}^K$. Also, if $T_{w(0)}, T_{w(1)}, \ldots, T_{w(2^K-1)}$ are empty ($n = |T_{w(0)}| + |T_{w(1)}| + \ldots + |T_{w(2^K-1)}| = 0$) then T' is empty and hence random, thus the first part of the lemma trivially holds for the basis of the induction.

Let us consider the case where $n > 0$, assuming now that both parts of the lemma are true for all sizes smaller than n.
We start with the split process and we denote by n the size of T, the quad tree to be partitioned. Let the root of T be y and let $t_{u(0)}, t_{u(1)}, \ldots, t_{u(2^K-1)}$ denote its subtrees.
If $y \prec_{u(i)} x$, then y is the root of the tree $T_{u(i)}$ and $t_{u(i)}$ its $u(i)$-th subtree. To complete $T_{u(i)}$ we apply recursively the split algorithm to the subtrees $t_{u(0)}, t_{u(1)}, \ldots, t_{u(i-1)}, t_{u(i+1)}, \ldots, t_{u(2^K-1)}$. Since $|t_{u(i)}| < n$, by the inductive hypothesis, we obtain the $2^K - 1$ random and independent subtrees of $T_{u(i)}$. Moreover, these subtrees are independent of $t_{u(i)}$ since they are obtained from the subtrees $t_{u(0)}, t_{u(1)}, \ldots, t_{u(i-1)}, t_{u(i+1)}, \ldots, t_{u(2^K-1)}$, which, by hypothesis are independent of $t_{u(i)}$. Simultaneously, from subtrees $t_{u(0)}, t_{u(1)}, \ldots,$ $t_{u(i-1)}, t_{u(i+1)}, \ldots, t_{u(2^K-1)}$, we obtain $2^K - 1$ trees for each of the subtrees $T_{u(0)}, T_{u(1)}, \ldots, T_{u(i-1)}, T_{u(i+1)}, \ldots, T_{u(2^K-1)}$. These trees are of size smaller than n and by inductive hypothesis, they are random and independent, so, also by inductive hypothesis, the result of joining them is a random quad tree, and thus $T_{u(0)}, T_{u(1)}, \ldots, T_{u(2^K-1)}$ are all random and independent. To complete the proof for this case, we need only show that for every key z of $T_{u(i)}$ the probability that z is at the root of $T_{u(i)}$ is $\frac{1}{m}$, where m is the size of $T_{u(i)}$. Indeed,

$$\begin{aligned} \mathbb{P}\left[\{z \text{ is root of } T_{u(i)} \mid y \prec_{u(i)} x\}\right] &= \frac{\mathbb{P}\left[\{z \text{ is root of } T \text{ and } z \prec_{u(i)} x\}\right]}{\mathbb{P}\left[\{z \prec_{u(i)} x\}\right]} \\ &= \frac{1/n}{m/n} = \frac{1}{m}. \end{aligned}$$

Now we tackle the second part of the lemma and show that join preserves randomness when $n > 0$. If exactly one of the trees $T_{w(0)}, T_{w(1)}, \ldots, T_{w(2^K-1)}$ is

not empty, then join returns the non-empty tree which, by hypothesis, is random. We consider now the case where at least two of the trees $T_{w(0)}, T_{w(1)}, \ldots,$ $T_{w(2^K-1)}$ are not empty. Let $T_{w(i)}$ have root y_i, and subtrees $t^i_{w(j)}$, (for all $j \in \{0, 1, \ldots, 2^K - 1\}$). If we select the key y_i to become the root of T', then we will recursively join $t^i_{w(j)}$ with the corresponding trees that result from splitting the trees $T_{w(0)}, T_{w(1)}, \ldots, T_{w(i-1)}, T_{w(i+1)}, \ldots, T_{w(2^K-1)}$ with respect to y_i. By the inductive hypothesis, from this splitting process result independent and random trees of sizes smaller than n. Thus, again by the inductive hypothesis joining some of them together produces a random quad tree. Thus, we have that the $w(i)$-th subtree of T' is $t^i_{w(i)}$, which is a random quad tree and that the other subtrees are also random since they are the result of splitting and joining subtrees of size smaller than n. All the subtrees are independent of each other and the probability that y_i was the root of $T_{w(i)}$ times the probability that it is selected as root of T' is $\frac{1}{|T_{w(i)}|} \times \frac{|T_{w(i)}|}{n} = \frac{1}{n}$. □

Theorem 4.3.2. *If T is a random quad tree that contains the set of compatible keys X and x is any key compatible with X, then* insert(T, x) *returns the random quad tree containing the set of keys $X \cup \{x\}$.*

Proof. By induction on the size n of T. If $n = 0$, then T is the empty tree (a random quad tree), and insert(T, x) returns a random quad tree with root x, and 2^K empty subtrees. We assume now that T is not empty and that the theorem is true for all sizes smaller than n. The insertion of x in T has two possible results, with probability $\frac{1}{(n+1)}$, x is the root of $T' =$ insert(T, x) and with complementary probability x is recursively inserted in the corresponding $w(i)$-th subtree of T.

Let us consider first the case in which x is not inserted at the root of T. Consider an item $y \in T$. The probability that y is the root of T before the insertion of x is $\frac{1}{n}$, since by hypothesis T is a random quad tree. The probability that y is at the root of T' is the probability that x is not at the root of T' and y was at the root of T, which is $\frac{1}{n} \times \frac{n}{n+1}$, resulting in the desired probability. Moreover, since x is not inserted at the root of T' in the first step, the insertion proceeds recursively in one of the subtrees of T, which are independent random quad trees of sizes smaller than n. Thus, T' is a random quad tree of size $n + 1$.

Finally, with probability $\frac{1}{n+1}$, x is the root of T'. The tree T, which by hypothesis is a random quad tree, must be split with respect to x, and

because of Lemma 4.3.1 this step produces 2^K independent random quad trees which are the subtrees of T', thus T' is also a random quad tree. □

Theorem 4.3.3. *If T is a random quad tree that contains the set of keys X, then,* delete(T, x) *produces a random quad tree T' that contains the set of keys $X \backslash \{x\}$.*

Proof. If the key x is not in T, then the algorithm does not modify the tree, which is random. Let us now suppose that x is in T, this case is proved by induction on the size of the tree. If $n = 1$, x is the only key in the tree and after deletion we obtain the empty tree which by definition is a random quad tree. We now assume that $n > 1$ and that the theorem is true for all sizes smaller than n. If x was not the key at the root of T we proceed recursively in one of the 2^K subtrees, and by inductive hypothesis we obtain a randomly built subtree. If x was the key at the root of T, the tree after deletion is the result of joining the 2^K subtrees of T, which produce a random quad tree because of Lemma 4.3.1. Finally, after deletion, each node has probability $\frac{1}{(n-1)}$ of being the root. Let y be any key in T such that $y \neq x$, then

$$\begin{aligned}
\mathbb{P}\left[\{y \text{ is the root of } T'\}\right] = \; & \mathbb{P}\left[\{y \text{ is the root of } T' \,|\, x \text{ was not the root of } T\}\right] \\
& \times \mathbb{P}\left[\{x \text{ was not the root of } T\}\right] \\
& + \mathbb{P}\left[\{y \text{ is the root of } T' \,|\, x \text{ was the root of } T\}\right] \\
& \times \mathbb{P}\left[\{x \text{ was the root of } T\}\right] \\
= \; & \tfrac{1}{n-1} \times \tfrac{n-1}{n} + \tfrac{1}{n-1} \times \tfrac{1}{n} \\
= \; & \tfrac{1}{n-1}
\end{aligned}$$

Thus, we obtain the desired probability. □

Combining the two previous theorems we obtain the following important corollary.

Corollary 4.3.4. *The result of any arbitrary sequence of randomized insertions and deletions, starting from an initially empty tree is always a random quad tree.*

Several random variables over random quad trees, which hold also for randomized quad trees, have been studied in the literature. For instance, the expected depth of insertion D_n of the n-th multidimensional key is in probability asymptotic to $(2/K) \log n$ [25]. The expected height H_n of a K-dimensional quad tree of size n is in probability asymptotic to $(c/K) \log n$,

where $c = 4.31107\ldots$ [23]. The cost of a random search has logarithmic mean and variance and is asymptotically distributed as a normal variable [36]. The cost of a partial match query with s coordinates specified is $\Theta(n^{1-s/K+\theta(s/K)})$, where the function $\theta(x)$ is defined as the solution $\theta \in [0,1]$ of the equation $(\theta+3-x)^x(\theta+2-x)^{1-x}-2=0$ [35].

The deletion algorithm here presented is simpler than the standard one and scales smoothly for larger dimensions, even though quad trees are not generally used for dimensions $K > 3$ because of their large space requirements.

5. FINGERED MULTIDIMENSIONAL TREES

In this chapter we propose schemes to augment standard and relaxed K-d trees[1] with fingers (Section 5.1) to improve the efficiency of orthogonal range and nearest neighbor searches which exhibit locality of reference (Section 5.2). Our experiments show that the second, more complex scheme of m-finger K-d trees exploits better the locality of reference than the simpler 1-finger K-d trees; however these gains do not compensate for the amount of memory and CPU time that m-finger trees require, so that 1-finger K-d trees are more attractive on a practical ground. The results of this chapter have been published in the paper *Fingered Multidimensional Search Trees* [31].

5.1 Finger K-d Trees

In this section we introduce two different schemes of fingered K-d trees. We call the first and simpler scheme 1-finger K-d trees; we augment the data structure with one finger pointer. The second scheme is called multiple finger K-d tree (or m-finger K-d tree, for short). Each node of the new data structure is equipped with two additional pointers or fingers, each pointing to descendent nodes in the left and right subtrees, respectively. The search in this case proceeds by recursively using the fingers whenever possible.

5.1.1 One-Finger K-d Trees

Definition 5.1.1. A *one-finger K-d tree* (1-finger K-d tree) for a set F of K-dimensional data points is a K-d tree in which:

1. Each node contains its bounding box and a pointer to its parent;
2. There is a pointer called *finger* that points to an arbitrary node of the tree.

The finger is initially pointing to the root but it is updated after each individual search.

We consider first orthogonal range searches. Range searching in relaxed and standard K-d trees is straightforward. We refer the reader to Algorithm 1 in Section 2.2. Taking, for instance, an orthogonal range query

[1] They actually apply to any variant of K-d trees, not just the two mentioned; some additional but minor modifications would be necessary to adapt them to quad trees, K-d tries, etc.

$Q = [0.451, 0.813] \times [0.285, 0.309]$ and the tree of Figure 3.1 in previous chapter, the range search algorithm will visit nodes x_1, x_2, x_5, x_6, x_9, and x_{10}.

In order to give the descriptions and some properties of the algorithms of this chapter we will require a lemma that relates the notion of bounding hyper-rectangles (given in Definition 2.2.3) with the range search algorithm for relaxed and standard K-d trees.

Lemma 5.1.1. ([14, 24]) *A point x with bounding hyper-rectangle $B(x)$ is visited by the range search algorithm with query hyper-rectangle Q if and only if $B(x)$ intersects Q.*

To illustrate this lemma, let us come back to the behavior of the range search algorithm with the range query $Q = [0.451, 0.813] \times [0.285, 0.309]$ in the K-d tree of Figure 3.1. The root x_1 is visited and its bounding hyper-rectangle, which is the whole search space, intersects Q. Since Q is completely below the second attribute of x_1, the search continues in the left subtree of x_1. But the second attribute of x_1 is a lower bound for the bounding hyper-rectangles of all the records that pertain to the right subtree of x_1 and it is therefore impossible that any of these bounding hyper-rectangles intersect Q. The bounding hyper-rectangle of x_2 is $B(x_2) = [0, 1] \times [0, 0.703]$ and it clearly intersects Q. Following the same reasoning as before, it is easy to see that it is impossible that Q intersects $B(x_7)$. However, Q intersects $B(x_5)$, $B(x_6)$, $B(x_9)$ and $B(x_{10})$ and the range search algorithm visits those nodes.

For 1-finger K-d trees the orthogonal range search algorithm starts the search at some node x pointed to by finger F. Let $B(x)$ be the bounding box of node x and Q the range query. If $Q \subset B(x)$ then (because of Lemma 5.1.1), all the points to be reported must necessarily be in the subtree rooted at x. Thus, the search algorithm proceeds from x down following the classical range search algorithm. Otherwise, some of the points that are inside the query Q can be stored in nodes which are not descendants of x. Hence, in this situation the algorithm backtracks until it finds the first ancestor y of x such that $B(y)$ completely contains Q. Once y has been found the search proceeds as in the previous case. The finger is updated to point to the first node where the range search must follow recursively into both subtrees (or to the last visited node if no such node exists). In other terms, F is updated to point to the node whose bounding box completely contains Q and none of the bounding boxes of its descendants does. The idea is that if consecutive queries Q and Q' are close in geometric terms then either the bounding box $B(x)$ that contains Q does also contain Q' or only a limited amount of

backtrack suffices to find the appropriate ancestor y to go on with the usual range searching procedure. Of course, the finger is initialized to point to the tree's root before the first search is made. Algorithm 13, herewith, describes the orthogonal range search in 1-finger K-d trees. It invokes the standard `range_search` algorithm once the appropriate starting point has been found.

For simplicity, the algorithm assumes that each node stores its bounding box; this implementation requires $\Theta(n)$ additional memory for the parent pointer, the finger pointer and bounding boxes, that is, a total of $n + 1$ additional pointers and $2n$ K-dimensional points. However, it is possible to modify the algorithm so that only the nodes in the path from the root to F contain this information or to use an auxiliary stack to store the bounding boxes of the nodes in the path from the root to the finger. Additionally, the explicit pointers to the parent can be avoided using pointer reversal plus a pointer to finger's parent or using the same stack that stores the bounding boxes in order to recover the followed path. This codification uses in average $\Theta(\log n)$ additional memory[2].

Algorithm 13 The orthogonal range search algorithm for 1-finger K-d trees.

```
▷ F: 1-finger K-d tree, Q: query, S: set of keys
function one_finger_range_search(F, Q, S) : 1-finger K-d tree
    if (F = nil) then return F;
    B := F → bounding_box;
    if (Q ⊄ B) then return one_finger_range_search(F → parent, Q, S);
    x := f → info;
    j := f → discr;
    if (Q.u[j] < x[j]) then return one_finger_range_search(F → left, Q, S);
    if (Q.l[j] ≥ x[j]) then return one_finger_range_search(F → right, Q, S);
    if (x ∈ Q) then
      S := S ∪ {x}
      range_search(F → left, Q, S);
      range_search(F → right, Q, S);
      return F;
end
```

Let $G(T, Q)$ be the node $x \in T$ such that $B(x) \supseteq Q$ and no other bounding box of a descendant of x contains Q (the range query). In other words, $B(x)$ is the minimal bounding box that completely contains Q. Ide-

[2] Actually, the necessary additional space is proportional to the height of the K-d tree, which on average is $\Theta(\log n)$ but can be as much $\Theta(n)$ in the worst-case.

ally $G(T,Q)$ is the node to point to with the finger, since in this case the overwork of the search algorithm is minimal. All the nodes that are from $G(T,Q)$ down in the tree with bounding box intersecting Q must be visited by Algorithm 13 in order to assure its correctness. In fact these nodes are also visited by the classical range search algorithm (by Lemma 5.1.1). We call these nodes hits. More precisely, we say that a node $x \in T$ is a *hit* (H) if it is a descendant of $G(T,Q)$ such that $B(x) \cap Q \neq \emptyset$. We call the rest of descendants of $G(T,Q)$ together with the rest of the nodes visited by Algorithm 13 *misses* (because they are visited when they should'nt) and we divide them in three groups: upper, lower and out-bounds misses. We say that a node $x \in T$ is an *upper miss* (UM) if it is an ancestor of $G(T,Q)$. We say that a node $x \in T$ is a *lower miss* (LM) if it is a descendant of $G(T,Q)$ such that $B(x) \cap Q = \emptyset$. Let $U(T,Q,F)$ be the node $x \in T$ with minimum bounding box such that x is an ancestor of F (the finger) and $B(x) \supseteq Q$. We say that a node $x \in T$ is an *out-bounds miss* (OM) if it is in the path from F to $U(T,Q,F)$ when F is neither descendant nor ancestor of $G(T,Q)$. Note that F can be $U(T,Q,F)$ if $B(F) \supseteq Q$, and that in this case there is no backtrack in the algorithm. Note also that $U(T,Q,F)$ is either an ancestor of $G(T,Q)$ or $G(T,Q)$. If F is an ancestor of $G(T,Q)$ then it coincides with $U(T,Q,F)$. If F points to $G(T,Q)$ or to any of its descendants then $U(T,Q,F)$ coincides with $G(T,Q)$. Finally, if F is not related to $G(T,Q)$ then $U(T,Q,F)$ is the first common ancestor of F and $G(T,Q)$. Figure 5.1 illustrates these definitions. The range search algorithm over 1-finger K-d trees visits then,

- The nodes in the path from F to $U(T,Q,F)$ (out-bounds or lower misses),
- The nodes in the path from $U(T,Q,F)$ to $G(T,Q)$ (upper misses),
- And the descendants of $G(T,Q)$ whose bounding box intersects Q (hits).

The nodes visited by Algorithm 13 are formally characterized by Lemma 5.1.2.

Lemma 5.1.2. *Let T be a one finger relaxed Kd-tree with finger F.*

1. *If F is an UM then a node $y \in T$ is visited by the range search algorithm if and only if y is a descendant of F and $Q \cap B(y) \neq \emptyset$.*

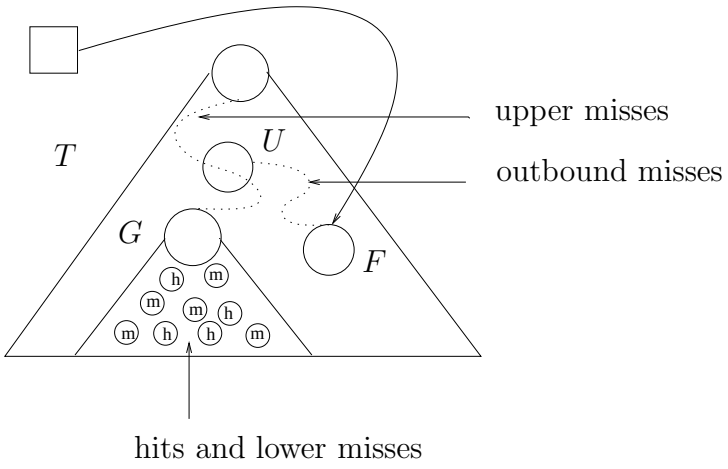


Fig. 5.1: Special nodes of a 1-finger K-d tree

2. *If F is a H or a LM then a node $y \in T$ is visited by the range search algorithm if and only if y is a descendant of $G(T, Q)$ such that y is in the path from F to $G(T, Q)$ or $Q \cap B(y) \neq \emptyset$.*

3. *If F is an OM then a node $y \in T$ is visited by the range search algorithm if and only if y is in the path from F to $U(T, Q, F)$ or y is a descendant of $U(T, Q, F)$ such that $Q \cap B(y) \neq \emptyset$.*

Proof. With respect to the proof of (1), we must observe that if finger F is an UM then it is in the path from the root of the tree to node $G(T, Q)$ and by definition, Algorithm 13 visits the tree starting at node F and following down with the non-fingered range search algorithm. Thus, by Lemma 5.1.1, (1) follows.
Let us suppose now that F is a H or a LM. In this case, the algorithm backtracks from F to $G(T, Q)$ and then follows down with the non-fingered range search algorithm. It follows that a node is visited if and only if it is visited during the backtrack (which means that it is in the path from F to $G(T, Q)$) or it is visited from $G(T, Q)$ down (which means, because of Lemma 5.1.1, that the intersection of Q and the bounding hyper-rectangle of the visited node is not empty). Henceforth, (2) follows.
Supposing that F is an OM, the algorithm backtracks from F to $U(T, Q, F)$ and then descends using the non-fingered range search algorithm. A node is

visited if and only if it is the path from F to $U(T, Q, F)$ or it is in the path of the range search algorithm from $U(T, Q, F)$ down. By Lemma 5.1.1, the nodes in this path are visited if and only if their bounding hyper-rectangles intersect Q. Hence, (3) follows. □

The single finger is exploited for nearest neighbor searches much in the same vein. Let be q the nearest neighbor query and let x be the node pointed to by finger F. Initially F will point to the root of the tree, but on successive searches it will point to the last closest point reported. The first step of the algorithm is then to calculate the distance d between x and q and to determine the ball with center q and radius d. If this ball is completely included in the bounding box of x then the nearest neighbor search algorithm proceeds down the tree exactly in the same way as the standard nearest neighbor search algorithm. If, on the contrary, the ball is not included in $B(x)$, the algorithm backtracks until it finds the least ancestor y whose bounding box completely contains the ball. Then the algorithm continues as the standard nearest neighbor search. Algorithms 14 and 15 describe the nearest neighbor algorithm for 1-finger K-d trees; notice that they behave just as the standard nearest neighbor search once the appropriate node where to start has been found.

Algorithm 14 The nearest neighbor search algorithm for 1-finger K-d trees.

```
▷ Precondition: F is not empty
function nearest(F : 1-finger K-d tree, q : query) : key
    one_finger_NN(F, q, ∞, F);
    return F → key;
end
```

5.1.2 Multiple Finger K-d Trees

Definition 5.1.2. A *multiple-finger K-d tree* (m-finger K-d tree) for a set F of K-dimensional data points is a K-d tree in which

- each node contains its bounding box, a pointer to its parent and
- two pointers, `fleft` and `fright`, pointing to two arbitrary nodes in its left and right subtrees, respectively.

Algorithm 15 The procedure `one_finger_NN` for the nearest neighbor search algorithm for 1-finger K-d trees.

```
▷ F: 1-finger K-d tree, q: query, min_dist: distance, nn: 1-finger K-d tree
procedure one_finger_NN(F, q, min_dist, nn) : 1-finger K-d tree
    x := F → info;
    d := dist(q, x);
    B := F → bounding_box;
    if d < min_dist then
      min_dist := d;
      nn := F;
    if (BALL(q, min_dist) ⊄ B) then ▷ Backtrack
      one_finger_NN(F → parent, q, min_dist, nn);
    j := F → discr;
    if (q[j] < x[j]) then
      one_finger_NN(F → left, q, min_dist, nn);
      other := F → right;
    else
      one_finger_NN(F → right, q, min_dist, nn);
      other := F → left;
    if (q[j] − min_dist ≤ x[j] and q[j] + min_dist ≥ x[j]) then
      one_finger_NN(other, q, min_dist, nn);
end
```

Given a m-finger K-d tree T and an orthogonal range query Q the orthogonal range search in T returns the points in T which fall inside Q as usual, but it also modifies the finger pointers of the nodes in T to improve the response time of future orthogonal range searches. The algorithm for m-finger search trees follows by recursively applying the 1-finger K-d tree scheme at each stage of the orthogonal range search trees. The fingers of visited nodes are updated as the search proceeds; we have considered that if a search continues in just one subtree of the current node the finger corresponding to the non-visited subtree should be reset, because it was not providing useful information. The pseudo-code for this algorithm is given as Algorithm 16.

The implementation of m-finger search trees does require $\Theta(n)$ additional memory for the parent pointer, finger pointers and bounding boxes, that is, a total of $3n$ additional pointers and $2n$ K-dimensional points. This could be a high price for the improvement in search performance which, perhaps, might not be worth paying.

The next lemmas state few characteristics of the behavior of the m-finger range search algorithm.

Lemma 5.1.3. *Let T be a m-finger K-d tree and Q an orthogonal range query. If a node $y \in T$ is a descendant of $G(T, Q)$ such that $Q \cap B(y) \neq \emptyset$ then y is visited by the m-finger range search algorithm.*

Proof. The proof is immediate from Lemma 5.1.1 because the m-finger range search algorithm allow not to visit nodes in the path from the root to $G(T, Q)$ that are visited by the non-fingered range search algorithm, but it should visit at least the same nodes below $G(T, Q)$ that visits the range search algorithm. □

Lemma 5.1.4. *If a node y is visited by the m-finger range search algorithm (without backtrack), then $B(y) \cap Q_i \neq \emptyset$ for some Q_i in a sequence of N queries, for $0 \leq i \leq N$.*

Proof. If a node y is visited by the m-finger range search algorithm and it is no visited by backtrack then it is visited either because it is a descendant of $G(T, Q)$ such that Q intersects $B(y)$ or because it is pointed by at least one of the fingers of some other node. In the first case, the lemma follows immediately; in the second, if y is pointed by any finger it means that it was the node were the algorithm followed the search visiting the two corresponding subtrees, at any previous stage. By definition the range query processed at that time was intersected by the bounding box of such a node, and the lemma follows. □

The multiple finger is not defined for nearest neighbor search. The reason is that while the multiple finger algorithm for orthogonal range search is based on successive decompositions of the (range) query, in the case of nearest neighbor we did not find a suitable way to exploit the decomposition of the sphere delimited by the nearest neighbor query and its closest (so far encountered) neighbor.

5.2 Locality Models and Experimental Results

Both 1-finger and m-finger K-d trees try to exploit locality of reference in long sequences of queries, so one of the main aspects of this chapter was to

devise meaningful models on which we could carry out the experiments. The programs used in the experiments described in this section have been written in C, using the GNU compiler `gcc-2.95.4`. The experiments themselves have been run in a computer with Intel Pentium 4 CPU at 2.8 GHz with 1 Gb of RAM and 512 Kb of cache memory.

5.2.1 The Models

In the case of orthogonal range search, given a size n, and a dimension K, we generate $T = 1000$ sets of n K-dimensional points drawn uniformly and independently at random in $[0,1]^K$. Each point of each set is inserted into two initially empty trees, so that we get a random standard K-d tree T_s and a random relaxed K-d tree T_r of size n which contain the same information. For each pair (T_s, T_r), we generate $S = 300$ sequences of $Q = 100$ orthogonal range queries and make the corresponding search with the standard and the fingered variants of the algorithm, collecting the basic statistics on the performance of the search.

We have performed experiments with fixed size and variable size queries, with up to $n = 50000$ elements per tree. For fixed size queries the length of the K edges of each query was $\Delta = 0.01$. For variable size queries the length of the edges was dependent of the number of nodes in the tree. The length of the K edges was $\Delta = c \cdot \sqrt[K]{1/n}$ and we performed experiments for $c = 1$, $c = 10$ and $c = 100$. To model locality, we introduced the notion of δ-close queries. We propose two different models: the first one relative to the length size side of the query, the second one independent of the query size.

Definition 5.2.1. Given to orthogonal range queries Q and Q' with identical edge lengths $\Delta_0, \Delta_1, \ldots, \Delta_{K-1}$, we say that Q and Q' are *δ-close* in relative terms if their respective centers z and z' satisfy $z - z' = (d_0, d_1, \ldots, d_{K-1})$ and $|d_j| \leq \delta \cdot \Delta_j$, for any $0 \leq j < K$.

Definition 5.2.2. Given to orthogonal range queries Q and Q' with identical edge lengths $\Delta_0, \Delta_1, \ldots, \Delta_{K-1}$, we say that Q and Q' are *δ-close* in absolute terms if their respective centers z and z' satisfy $z - z' = (d_0, d_1, \ldots, d_{K-1})$ and $|d_j| \leq \delta$, for any $0 \leq j < K$.

The sequences of δ-close queries were easily generated by choosing the initial center z_0 uniformly at random in $[-\Delta/2, 1+\Delta/2]^K$ (see Chapter 8) and setting each successive center $z_{m+1} = z_m + d_m$ for some randomly generated

vector d_m; in particular, the i-th coordinate of d_m is generated uniformly at random in $[-\delta \cdot \Delta_j, \delta \cdot \Delta_j]$ in the relative model and in $[-\delta, \delta]$ in the absolute one.

The real-valued parameter δ is a simple way to capture into a single number the degree of locality of reference. For instance, in the relative model, if $\delta < 1$ then δ-close queries must overlap at least a fraction $(1-\delta)^K$ of their volume. When $\delta \to \infty$ (in fact it suffices to set $\delta = \max\{\Delta_i^{-1}\}$) there is no locality of reference.

For nearest neighbor searches, the experimental setup was pretty much the same as for orthogonal search; for each pair (T_s, T_r) of randomly built K-d trees, we perform nearest neighbor search on each of the $Q = 100$ queries of each of the $S = 300$ generated sequences. As for orthogonal range, we propose two models of locality: the relative model and the absolute model.

Definition 5.2.3. Successive nearest neighbor queries q and q' are said to be δ-close in relative terms if $q - q' = (d_0, d_1, \ldots, d_{K-1})$ and $|d_j| \leq \delta \cdot \sqrt[K]{1/n}$, for any $0 \leq j < K$.

Definition 5.2.4. Successive nearest neighbor queries q and q' are said to be δ-close in absolute terms if $q - q' = (d_0, d_1, \ldots, d_{K-1})$ and $|d_j| \leq \delta$, for any $0 \leq j < K$.

In absolute terms, nearest neighbor queries are δ-close if the L_∞-norm $\|q - q'\|_\infty$ is smaller than δ. As such, only values in the range $[0, \sqrt{K}]$ are meaningful, although we find convenient to say $\delta \to \infty$ to indicate that there is no locality of reference.

5.2.2 The Experiments

Range Queries

We show in this section only the results corresponding to relaxed K-d trees with the model of relative locality and fixed length queries. Since the results for variable length queries, standard K-d trees and absolute locality are qualitatively similar, we present them in Appendix A.

To facilitate the comparison between the standard algorithms and their fingered counterparts we use the ratio of the respective overworks; namely, if $W_n^{(1)}$ denotes the overwork of 1-finger search, $W_n^{(m)}$ denotes the overwork of m-finger search and $W_n^{(0)}$ denotes the overwork of standard search (no

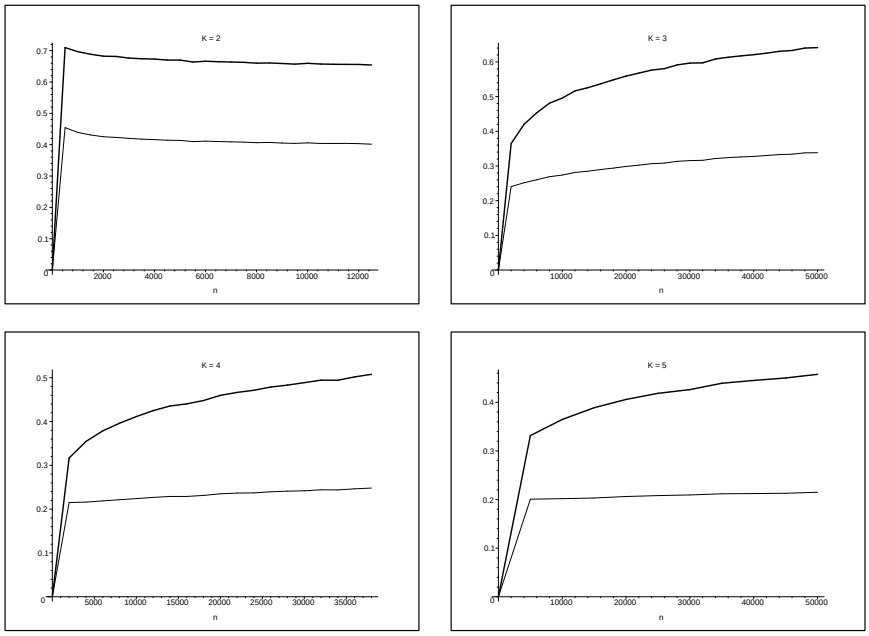


Fig. 5.2: Overwork ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.25$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

fingers), we will use the ratios $\tau_n^{(1)} = W_n^{(1)}/W_n^{(0)}$ and $\tau_n^{(m)} = W_n^{(m)}/W_n^{(0)}$. Recall that the overwork is the number of visited nodes during a search minus the number of nodes (points) which satisfied the range query. The graphs of Figures 5.2, 5.3 and 5.4 depict $\tau_n^{(1)}$ and $\tau_n^{(m)}$ for $\delta = 0.25$, $\delta = 0.75$ and $\delta = 2$ respectively.

All the plots confirm that significant savings in the number of visited nodes can be achieved thanks to the use of fingers; in particular, m-finger K-d trees do much better than 1-finger K-d trees for all values of K and δ. As δ increases the savings w.r.t. non-fingered search decrease, but even for $\delta = 2$ the overwork of 1-finger search is about 70% of the overwork of the standard search.

As we already expected, the performance of both 1-finger K-d trees and m-finger K-d trees heavily depends on the locality parameter δ, a fact that is well illustrated by Figures 5.5 and 5.6, that show the plot of the overwork $W_n^{(m)}$ of relaxed m-finger K-d trees for various values of δ and dimensions

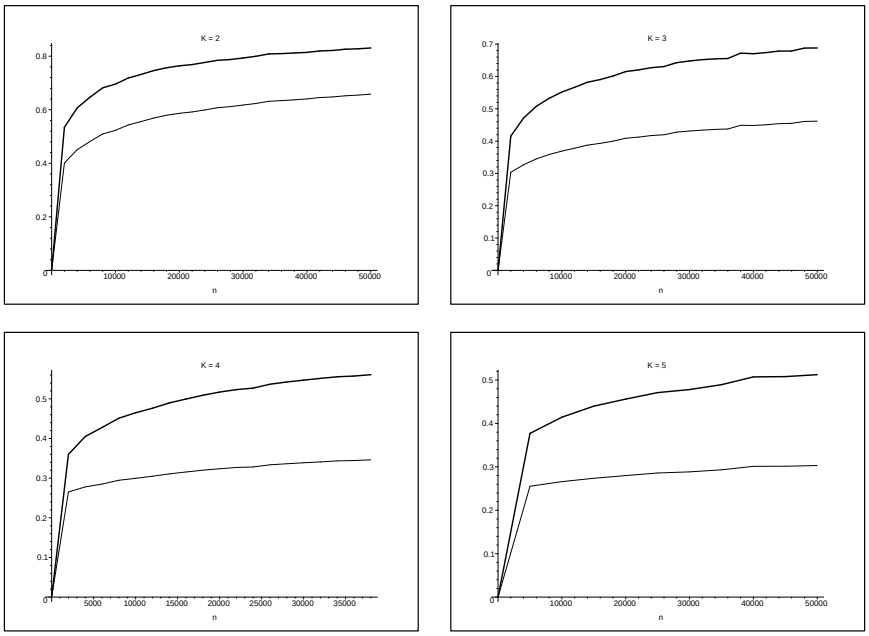


Fig. 5.3: Overwork ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.75$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

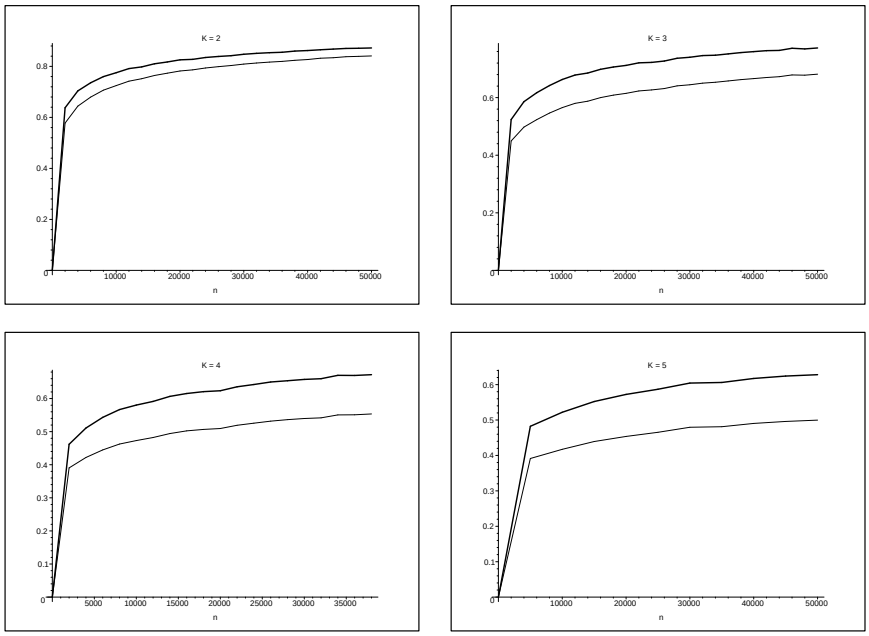


Fig. 5.4: Overwork ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 2$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

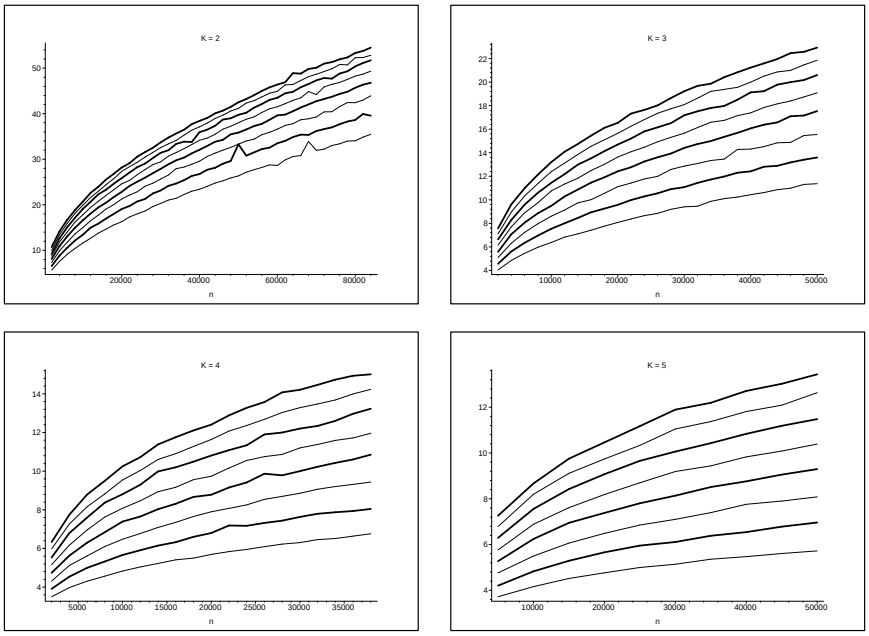


Fig. 5.5: Overwork in relaxed m-finger K-d trees for several values of the locality parameter δ, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*). In all graphs the values of δ are $0.25, 0.5, 0.75, \ldots, 2.00$ from bottom to top.

$2, 3, 4$ and 5 respectively. In particular, when the dimension increases we shall expect big differences in the savings that fingered search yield as δ varies; for lower dimensions, the variability of $W_n^{(m)}$ with δ is not so "steep". Similar phenomena can be observed for relaxed m-finger K-d trees and standard and relaxed 1-finger K-d trees. On the other hand, Figure 5.6 shows the variation of $\tau_{50000}^{(1)}$ and $\tau_{50000}^{(m)}$ as functions of δ for relaxed 1-finger and m-finger K-d trees.

Taking into account the way the algorithm works and the results of Chapter 8, we conjecture that 1-finger search reduces by a constant factor the logarithmic term in the overwork. Thus, if standard search has overwork $W_n^{(0)} \approx \sum_{0<j<K} \beta_j n^{\phi(j/K)} + \xi \log n$ then

$$W_n^{(1)} \approx \sum_{0<j<K} \beta_j n^{\phi(j/K)} + \xi' \log n, \tag{5.2.1}$$

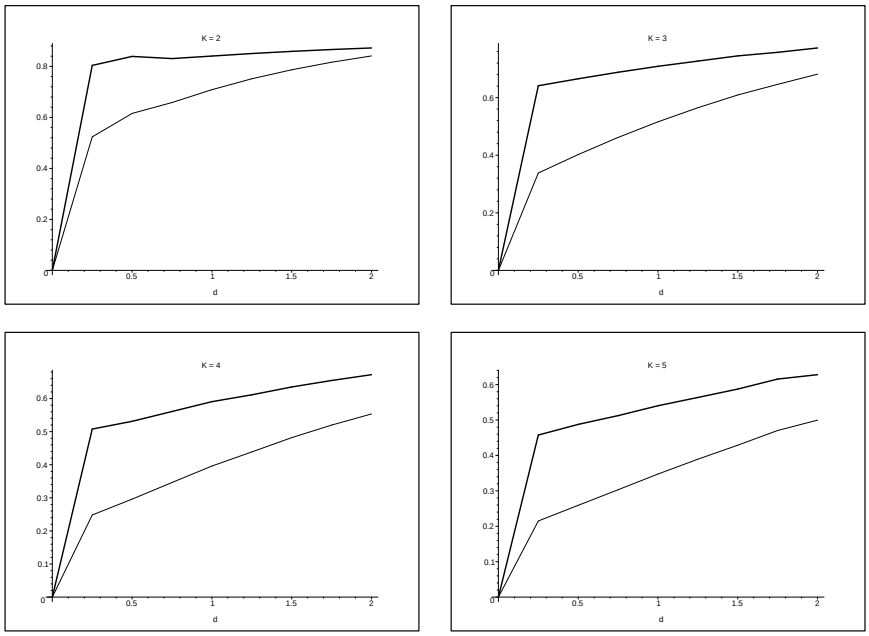


Fig. 5.6: Overwork ratios for $n = 50000$ for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

Tab. 5.1: Best-fit β and ξ for relaxed 2-d trees

δ	no finger		1-finger		m-finger	
	β	ξ	β	ξ	β	ξ
0.25	0.037	1.912	0.040	0.746	0.028	0.338
0.50	”	”	”	0.0836	0.031	0.450
0.75	”	”	”	0.902	0.034	0.550
1.00	”	”	”	0.955	0.035	0.647
1.25	”	”	”	0.999	0.037	0.733
1.50	”	”	”	1.054	0.038	0.824
1.75	”	”	”	1.103	0.039	0.908
2.00	”	”	”	1.151	0.039	0.986

with $\xi' = \xi'(\delta)$. However, since the ϕ's and β's are quite small it is rather difficult to disprove this hypothesis on an experimental basis; besides it is fully consistent with the results that we have obtained.

On the other hand, and again, following our intuitions on its *modus operandi*, we conjecture that the overwork of m-finger search is equivalent to skipping the initial logarithmic path and then performing a standard range search on a random tree whose size is a fraction of the total size, say n/x, for some $x > 1$ (basically, the m-finger search behaves as the standard search, but skips more or long intermediate chains of nodes and their subtrees). In other words, we would have

$$W_n^{(m)} \approx \sum_{0<j<K} \beta'_j n^{\phi(j/K)} + \xi' \log n, \tag{5.2.2}$$

for some β'_j's and ξ' which depend on δ (but ξ' here is not the same as for 1-finger search). In this case we face the same problems in order to find experimental evidence against the conjecture.

Table 5.1 summarizes the values of $\beta \equiv \beta_1$ and ξ that we obtain by finding the best-fit curve for the experimental results of relaxed 2-d trees. It is worth to recall here that the theoretical analysis in Chapter 8 predicts for the overwork $W_n^{(0)}$ of standard search in relaxed 2-d trees the following values: $\phi(1/2) = (\sqrt{5}-1)/2 \approx .618033989$, $\beta = 4\Delta(1-\Delta)\frac{\Gamma(2\alpha+1)}{(\alpha+1)\alpha^3\Gamma^3(\alpha)} \approx 0.03828681802$ and $\xi = 2(1-\Delta)^2 = 1.9602$.

For every experiment we count the number of nodes visited by the algorithms during backtrack. Since we want to know the amount of the overwork

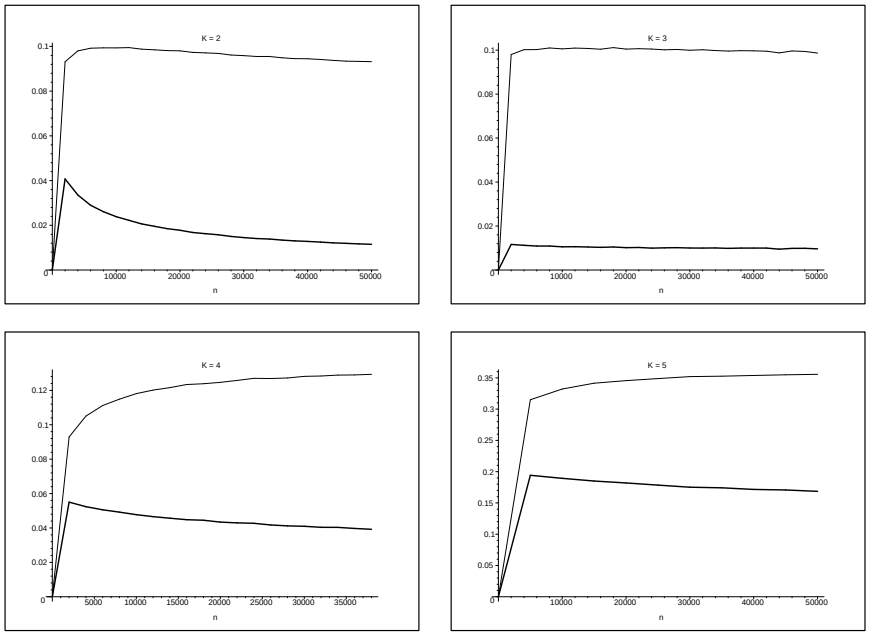


Fig. 5.7: Backtrack ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.25$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

that corresponds to backtrack, we take the respective ratios; that is, if $B_n^{(1)}$ denotes the backtrack of 1-finger search and $B_n^{(m)}$ denotes the backtrack of m-finger search, we will use the ratios $\kappa_n^{(1)} = B_n^{(1)}/W_n^{(1)}$ and $\kappa_n^{(m)} = B_n^{(m)}/W_n^{(m)}$. The graphs of Figures 5.7, 5.8, and 5.9 depict $\kappa_n^{(1)}$ and $\kappa_n^{(m)}$ for $\delta = 0.25$, $\delta = 0.75$ and $\delta = 2$ respectively.

As expected, the amount of backtrack for relaxed m-finger K-d trees is higher than this ratio for relaxed 1-finger K-d trees and in both cases this amount augments as the locality parameter and the dimension increase. It is worth to observe that for fixed dimensions the amount of backtrack ratio tends to be constant. This fact might reflect that the backtrack is of the same order of growth than the overwork for both 1-finger and m-finger K-d trees.

Concerning the time of CPU, it turns out that the total amount of CPU time required to answer the sequences of proposed queries in relaxed 1-finger K-d trees is less than the one required for plain relaxed K-d trees which in

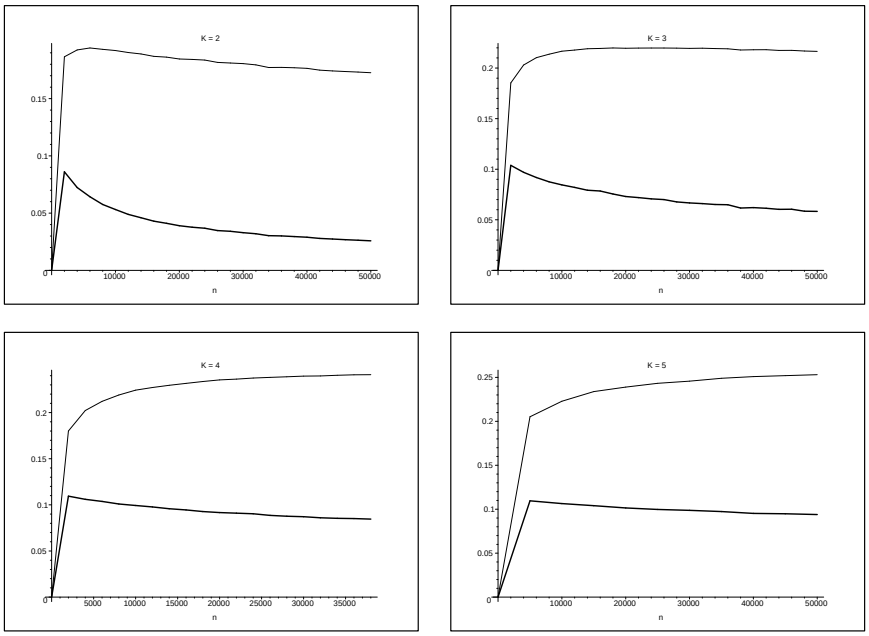


Fig. 5.8: Backtrack ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.75$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

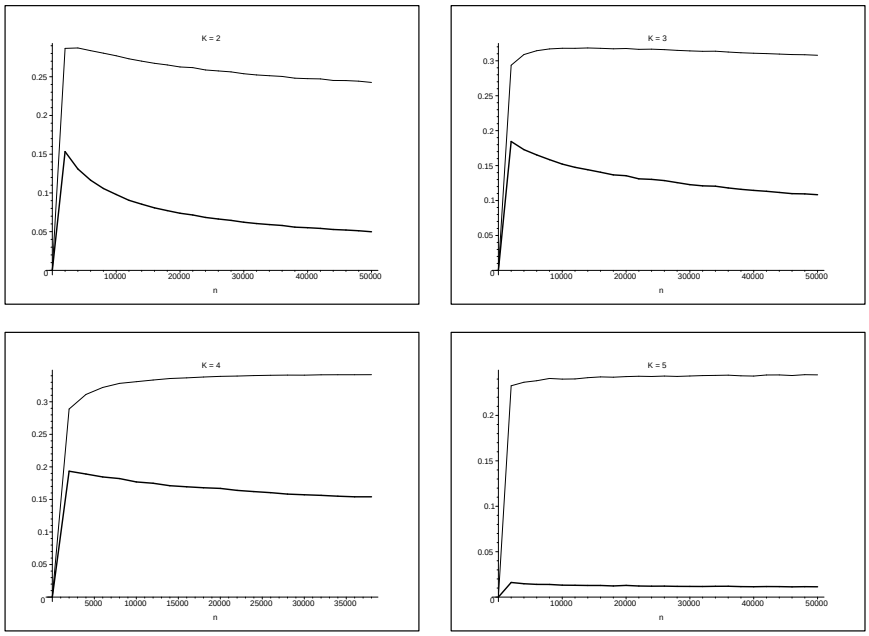


Fig. 5.9: Backtrack ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 2$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

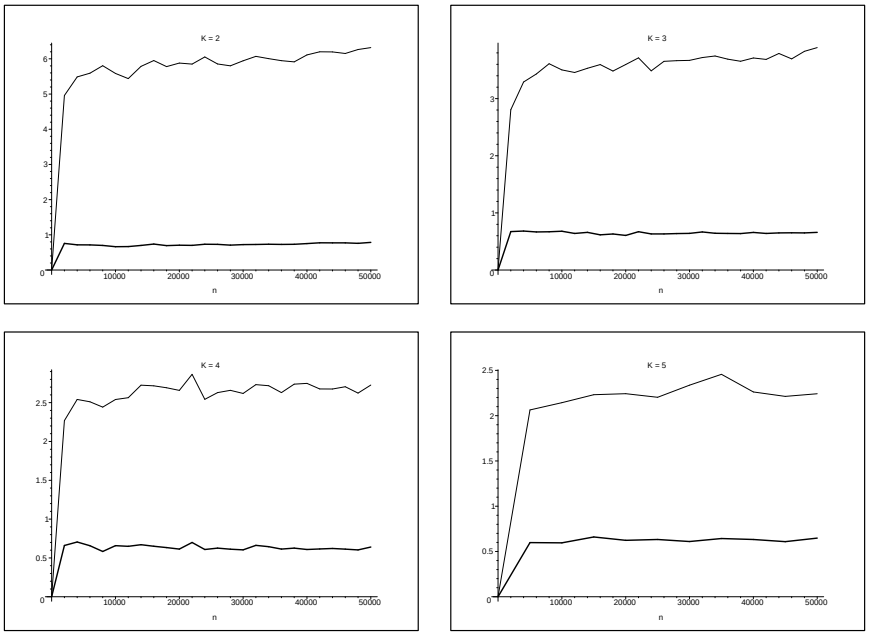


Fig. 5.10: Time ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.25$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

turn is much smaller than the one for m-finger K-d trees.

Let $C_n^{(1)}$ denote the total time of 1-finger search, $C_n^{(m)}$ denote the total time of m-finger search and, $C_n^{(0)}$ denote the total time of non-finger search, we will use the ratios $\zeta_n^{(1)} = C_n^{(1)}/C_n^{(0)}$ and $\zeta_n^{(m)} = C_n^{(m)}/C_n^{(0)}$. The graphs of Figure 5.10 depict $\zeta_n^{(1)}$ and $\zeta_n^{(m)}$ for $\delta = 0.25$. These experiments suggest that m-finger K-d trees are not a good alternative in practice, since they incur too much overhead in memory space and CPU time. Intuitively, it seems that these huge requirements of CPU time are due to the high number of pointer updates that the range search algorithm performs at each recursive step.

Nearest Neighbor Queries

The curves in Figure 5.11 show the performance of relaxed 1-finger K-d trees when the absolute locality model is used. There, we plot the ratio of the cost

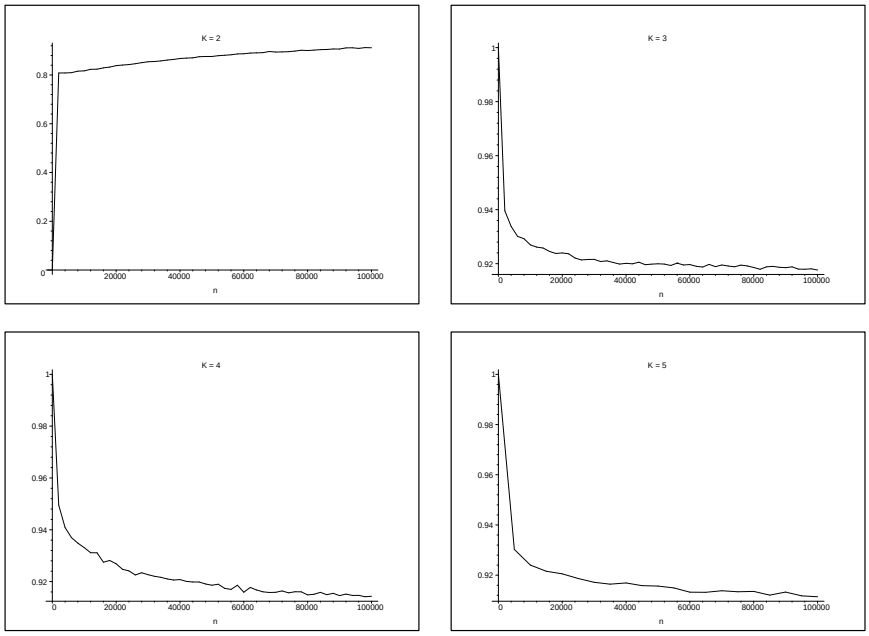


Fig. 5.11: Nearest neighbor queries in relaxed 1-finger K-d trees for $\delta = 0.005$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

using 1-finger nearest neighbor search to the cost using no fingers. For each dimension K ($K = 2, 3, 4, 5$), the curve corresponds to nearest neighbor search with $\delta = 0.005$.

It is not a surprise that when we have better locality of reference (a smaller δ) the performance improves. It is more difficult to explain why in this absolute model the variability on δ is smaller as the dimension increases. The qualitatively different behavior for $K = 2$, $K = 3$ and $K > 3$ is also surprising. For $K = 2$ the ratio of the costs increases as n increases until it reaches some stable value (e.g., roughly 90% when $\delta = 0.005$). For $K = 3$ we have rather different behavior when we pass from $\delta = 0.005$ to $\delta = 0.01$[3]. For $K = 4$, $K = 5$ and $K = 6$ we have the same qualitative behavior in all cases[4]: a decrease of the ratio as n grows until the ratio reaches a limit value. A similar phenomenon occurs for $K = 2$ and $K = 3$ provided that δ is even

[3] The plots for $\delta = 0.01$ are not shown in the figure.
[4] The plot for $K = 6$ is not shown in the figure.

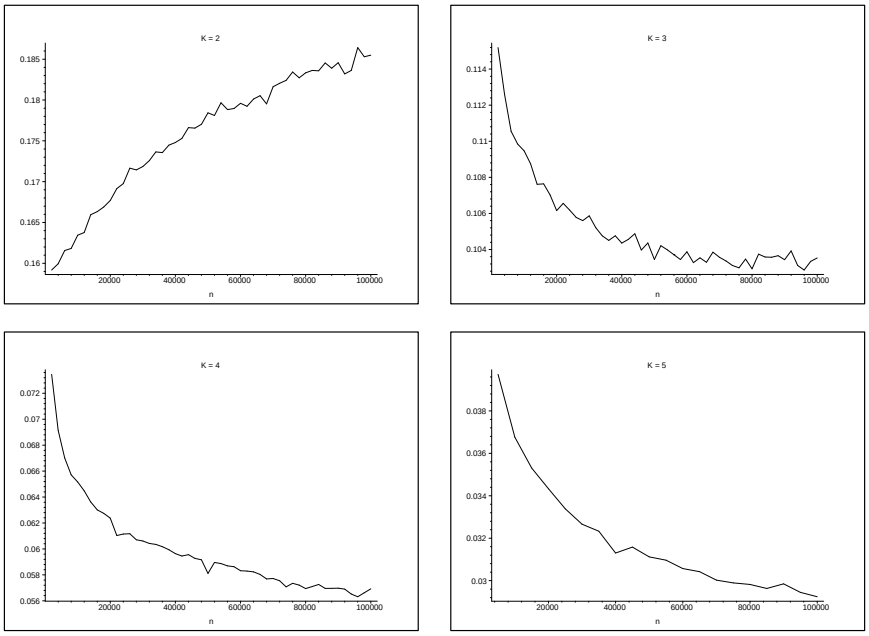


Fig. 5.12: Backtrack for nearest neighbor queries in relaxed 1-finger K-d trees for $\delta = 0.005$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

smaller than 0.005. In other words if $\delta \approx \sqrt[K]{1/n}$ or smaller then we will find the same qualitative behavior as in the case for $K = 3, 4, 5$; but if $\delta \gg \sqrt[K]{1/n}$ then we find a rather different qualitative behavior well represented here by the experiments for $K = 2$ (see also Figure 5.12).

The behavior of the backtrack is shown in the graphs of Figure 5.12. The plots represent the ratios of the number of nodes visited during backtrack divided by the number of nodes visited by the whole overwork. From the graphs, it can be observed that the amount of backtrack decreases as the dimension increases. For fixed dimension, the amount of backtrack seems to tend to a constant quantity. For the performed experiments, it goes from approximately a 20% for $K = 2$ to a 3% for $K = 5$.

The curves in Figure 5.13 show the performance of relaxed 1-finger K-d trees when the relative locality model is used. There, we plot the ratio of the cost using 1-finger nearest neighbor search to the cost using no fingers. For each dimension K ($K = 2, 3, 4, 5$), the curves correspond to nearest neighbor

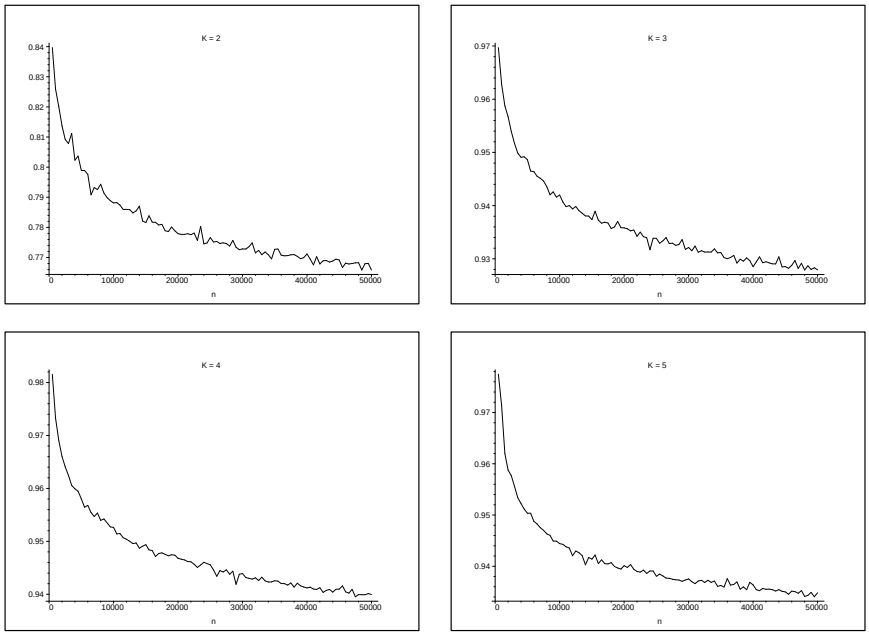


Fig. 5.13: Nearest neighbor queries in relaxed 1-finger K-d trees, $\delta = 0.25\sqrt[K]{1/n}$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

search with $\delta = 0.25\sqrt[K]{1/n}$. The behavior of nearest neighbor search for the relative locality model is completely different than for the absolute one. In this case, the gains decrease as augment the dimension, and for fixed dimension, they seem to tend to a constant value. For $K = 2$ it is observed a 30% of improvement and it tends to stabilize to a 5% when dimension increases.

Taking the number of visited nodes as measure of the cost of the nearest neighbor algorithm, we did not find significant improvements of 1-finger search with respect to standard search in none of our experiments, in particular, the cost of 1-finger nearest neighbor search was not below 90% of the standard cost even for large dimensions and small (but not unrealistic) δ's. However, it is worth to observe that the time of CPU required for 1-finger search is less than the time required for standard search. In this case, the improvements are significative (up to a 60% when $K = 2$). The improvements in CPU time decrease as the dimension increases (due to the *curse of*

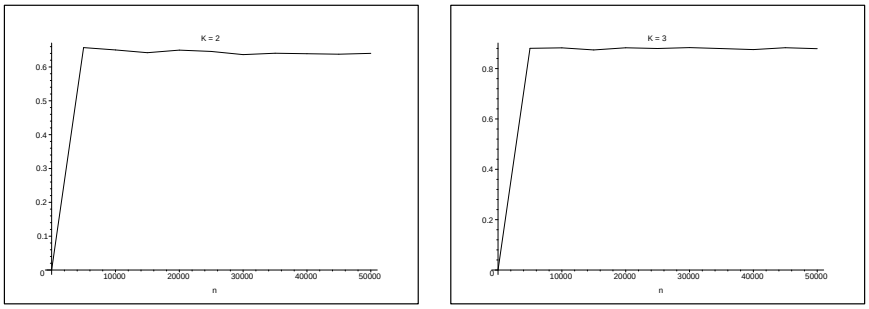


Fig. 5.14: Time ratios for nearest neighbor queries in relaxed 1-finger K-d trees, $\delta = 0.25\sqrt[K]{1/n}$, $K = 2$ (*left*), $K = 3$ (*right*).

dimensionality? [15]). Figure 5.14 shows the plots of the total CPU time for $K = 2$ and 3 when the relative locality model is used with $\delta = 0.25\sqrt[K]{1/n}$.

Algorithm 16 The orthogonal range search algorithm in a m-finger K-d tree.

```
▷ F: m-finger K-d tree, Q: query, S: set of keys
function multiple_finger_range_search(F, Q, S) : m-finger K-d tree
    if (F = nil) then return F;
    B := F → bounding_box;
    if (Q ⊄ B) then ▷ Backtrack
      F → fleft := F → left;
      F → fright := F → right;
      return multiple_finger_range_search(F → parent, Q, S);
    x := F → info;
    if (Q.u[j] < x[j]) then
      F → fright := F → right;
      F → fleft := multiple_finger_range_search(F → fleft, Q, S);
      if (F → fleft = nil) then return F;
      if (F → fleft = F) then F → fleft := F → left;
      return T → fleft;
    if (Q.l[j] ≥ x[j]) then
      F → fleft := F → left;
      F → fright := multiple_finger_range_search(F → fright, Q, S);
      if (F → fright = nil) then return F;
      if (F → fright = F) then F → fright := F → right;
      return T → fright;
    if (x ∈ Q) then S := S ∪ {x};
    F → fleft := multiple_finger_range_search(F → fleft, Q′, S);
    if (F → fleft = nil) then return F;
    if (F → fleft = F) then F → fleft := F → left;
    F → fright := multiple_finger_range_search(F → fright, Q″, S);
    if (F → fright = nil) then return F;
    if (F → fright = F) then F → fright := F → right;
    return F;
end
```

Part IV

ANALYSIS OF MULTIDIMENSIONAL DATA STRUCTURES

6. MATHEMATICAL PRELIMINARIES

6.1 Generating Functions

This chapter is devoted to the introduction of those mathematical tools that are required for the analysis of the algorithms presented in next chapters. We assume that the reader is familiar with concepts such as analyticity, convergence of power series, singularity and residue complex analysis and, Taylor series developments. Excellent references on these issues are [51, 52].

The basic tool in the average case analysis of algorithms and data structures used in this work is the *generating function* [44].

Definition 6.1.1. Given any sequence of complex numbers $\{a_k\}_{k\geq 0}$, its generating function is

$$A(z) = \sum_{k\geq 0} a_k z^k$$

where z is an auxiliary variable and k a non-negative integer.

The generating function of a sequence is often called *ordinary* generating function, in order to distinguish it from other kinds of generating functions such as *exponential* generating functions or *probability* generating functions.

Generating functions associate sequences of numbers to formal power series making possible to manipulate them with classical algebraic methods. In fact, elementary operations over sequences can be easily translated into operations over the corresponding generating functions. See Table 6.1, where operations (1), (3) and (4) correspond to *sum*, *backward shift* and *forward shift* of sequences. Operation (2) is known as *convolution* of sequences and operations (5) and (6) are *differentiation* and *integration* of sequences, respectively.

In many applications, it is often the case that the power series under study are convergent and in consequence they can be treated by analytical methods. In such cases the variable z of a generating function $f(z)$ is considered as a complex variable and the generating function as a complex function of z.

The n-th coefficient of a formal power series $f(z)$ will be denoted by $[z^n]f(z)$ (which also denotes the n-th coefficient of the Taylor expansion of an analytic function $f(z)$ around $z = 0$). Thus, if $f(z) = \sum_{n\geq 0} f_n z^n$ it follows that $[z^n]f(z) = f_n$.

In the average case analysis of algorithms it is necessary to recover the n-th coefficient of a generating function. There are two useful theorems that allow to extract this coefficient exactly, under particular circumstances.

Sequences	Generating Functions
1. $c_n = a_n \pm b_n$	$C(z) = A(z) \pm B(z)$
2. $c_n = \sum_{k=0}^{n} a_n b_{n-k}$	$C(z) = A(z) \cdot B(z)$
3. $c_n = a_{n-1}$	$C(z) = zA(z)$
4. $c_n = a_{n+1}$	$C(z) = \frac{A(z)-A(0)}{z}$
5. $c_n = na_n$	$C(z) = z\frac{d}{dz}A(z)$
6. $c_n = \frac{a_n}{n}$	$C(z) = \int_0^z (A(z) - A(0))\frac{dt}{t}$

Tab. 6.1: Translation of basic operations over sequences onto operations over generating functions.

The first one is the so-called general expansion theorem for rational generating functions while the second is the Lagrange inversion formula, that can be applied when the generating functions satisfy a specific kind of implicit equation. The statements are as follows.

Theorem 6.1.1. (Expansion Theorem for Distinct Roots) *If $R(z) = P(z)/Q(z)$, for some polynomials $P(z)$ and $Q(z)$, such that the degree of $P(z)$ is smaller than the degree of $Q(z)$, and $Q(z)$ has r distinct roots $\rho_1, \rho_2, \ldots, \rho_r$ of multiplicities $d_1, d_2, \ldots, d_r$, then*

$$[z^n]R(z) = f_1(n)\rho_1^n + f_2(n)\rho_2^n + \cdots + f_r(n)\rho_r^n, \qquad n \geq 0,$$

where each $f_k(n)$ is a polynomial of degree $d_k - 1$ and its leading coefficient a_k is

$$a_k = \frac{d_k P(1/\rho_k)(-\rho)^{d_k}}{Q^{(d_k)}(1/\rho_k)}.$$

A proof of this theorem can be found in [44], in page 340.

Theorem 6.1.2. (Lagrange Inversion Formula) *Let $\phi(u) = \sum_{n\geq 0} \phi_n z^n$ be a formal power series such that $\phi(0) = \phi_0 \neq 0$. Then the equation*

$$y(z) = z\phi(y(z))$$

has a unique formal power series solution that satisfies

$$y(z) = \sum_{n>0} \frac{z^n}{n}[y^{n-1}]\phi^n(y).$$

However, to find an exact expression for the coefficients of a power series is not always possible and a good enough solution is to get asymptotic estimates. In order to get asymptotic estimates of the n-th coefficient of generating functions there are useful methods provided by the analysis of function's singularities that we describe in next section.

6.2 Singularity Analysis

As we already said, generating functions can be considered as functions of complex variable in the complex plane, analytic in a disk around the origin. A *singularity* is a point at which the function ceases to be analytic. Let $f(z)$

be the generating function of a certain sequence. Since the power series of the function is analytic in the largest disk centered at the origin containing no singularities, the first step in the analysis will be to look for the singularities that are nearest to the origin. The nearest singularity is called *dominant singularity* and the distance from the origin to the dominant singularity is called *radius of convergence* of the power series. The radius of convergence provides useful information about the behavior of the coefficients of the power series, $f_n = [z^n]f(z)$, as stated by next theorem, which relates the location of singularities of a function to the exponential growth of its coefficients.

Theorem 6.2.1. (Exponential Growth Formula) *Let ρ be the radius of convergence of the power series $f(z) = \sum_{n\geq 0} f_n z^n$. Then, for all $\epsilon > 0$,*

$$(1-\epsilon)^n \rho^{-n} <_{i.o.} f_n <_{a.e.} (1+\epsilon)^n \rho^{-n},$$

where $<_{i.o.}$ means that the inequality holds for an infinite number of values of n, whereas $<_{a.e.}$ means that the inequality holds for all values of n, except a finite number of them.

Although these lower and upper bounds are useful information about the exponential growth of the coefficients f_n, it is usually insufficient and it is required to look for information on their sub-exponential growth, or preferable, to look for an asymptotic equivalent.

It is known from analysis that a non-entire function with positive coefficients has always a dominant positive real singularity. In most cases, it is possible to obtain information about the asymptotic behavior of the coefficients f_n by extracting information about the nature of the dominant singularity of the function $f(z)$ and the behavior of the function around it.

Singularity analysis methods are based on the assumption that a function $f(z)$ has, around its dominant singularity 1, an asymptotic expansion of the form: $f(z) = \sigma(z) + R(z)$ with $R(z) << \sigma(z)$ as $z \to 1$ and where $\sigma(z)$ is a standard set of functions that include $(1-z)^a \log^b(1-z)$ for constants a and b. Then, under general conditions,

$$[z^n]f(z) = [z^n]\sigma(z) + [z^n]R(z),$$

with $[z^n]R(z) << [z^n]\sigma(z)$, as $n \to \infty$.

Applications of this principle are based on varying the conditions imposed to functions $f(z)$ and $R(z)$ resulting in three principal methods:

1. *Transfer methods* where the approximation is established for $z \to 1$ and there are usually suppositions on the growth of the remainder term $R(z)$.

2. *Tauberian theorems* which impose conditions of positivity and monotonicity to be satisfied by the coefficients f_n and hold when z is real and less than 1. These theorems also require conditions on the growth of $R(z)$ but less restrictive than those of transfer methods.

3. *Darboux's method* imposes as condition the differentiability of $R(z)$.

Although in next chapter we will use Transfer-Lemma 6.2.2 and Corollaries 6.2.3 and 6.2.4, we also give for completion the expressions of Darboux's theorem and the Tauberian theorem of Hardy, Littlewood and Karamata.

Lemma 6.2.2. (Transfer Lemma [37]) *Assume that $f(z)$ is analytic in $|z| < 1$. Assume further that as $z \to 1$ in this domain,*

$$f(z) = \mathcal{O}\left(|1-z|^{\alpha}\right),$$

for some real number $\alpha < -1$. Then the n-th Taylor coefficient of $f(z)$ satisfies,

$$f_n = [z^n]f(z) = \mathcal{O}\left(n^{-\alpha-1}\right).$$

Corollary 6.2.3. *Assume that $f(z)$ is analytic in $|z| < 1$. Assume further that as $z \to 1$ in this domain,*

$$f(z) = o\left(|1-z|^{\alpha}\right),$$

for some real number $\alpha < -1$. Then the n-th Taylor coefficient of $f(z)$ satisfies,

$$f_n = [z^n]f(z) = o\left(n^{-\alpha-1}\right).$$

Before giving the statement of next corollary we require the definition of asymptotic equivalence. We say that the coefficients a_n and b_n are *asymptotically equivalents* and we denote it $a_n \sim b_n$, if and only if

$$\lim_{n\to\infty} \frac{a_n}{b_n} = 1.$$

Equivalently, we say that the functions $f(n)$ and $g(n)$ are *asymptotically equivalents* and we denote it $f(n) \sim g(n)$, if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$$

This is slightly more informative than Θ as it gives the coefficient of the leading term as well as the precise order.

Corollary 6.2.4. *Assume that $f(z)$ is analytic in $|z| < 1$. Assume further that as $z \to 1$ in this domain,*

$$f(z) \sim \left(|1-z|^{\alpha}\right),$$

for some real number $\alpha < -1$. Then the n-th Taylor coefficient of $f(z)$ satisfies,

$$f_n = [z^n]f(z) \sim \left(n^{-\alpha-1}\right),$$

Theorem 6.2.5. (Tauberian Theorem of Hardy-Littlewood-Karamata) *Assume that the function $f(z) = \sum_{n\geq 0} f_n z^n$ has radius of convergence 1 and satisfies for real z, $0 \leq z < 1$,*

$$f(z) \sim \frac{1}{(1-z)^s} L\left(\frac{1}{1-z}\right),$$

as $z \to 1^-$, where $s > 0$ and $L(u)$ is a function varying slowly at infinity. If $\{f_n\}_{n\geq 0}$ is monotonic, then

$$f_n \sim \frac{n^{s-1}}{\Gamma(s)} L(n),$$

where $\Gamma(z)$ denotes Euler's gamma function.

Theorem 6.2.6. (Darboux) *Let $f(z)$ be an analytic function in the open disk $|z| < \rho$ and assume that there is a unique singularity in the convergence circle at $z = \rho$. Furthermore, in a neighborhood of $z = \rho$, $f(z)$ satisfies*

$$f(z) = \left(1 - \frac{z}{\rho}\right)^{-\beta} g(z) + h(z),$$

for some analytic functions $g(z)$ and $h(z)$ at $z = \rho$, with $g(\rho) \neq 0$, and for some real $\beta \notin \{0, -1, -2, -3, \ldots\}$. Then,

$$[z^n]f(z) = f_n \sim \rho^{-n} n^{\beta-1} \frac{g(\rho)}{\Gamma(\beta)} \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right).$$

where $\Gamma(z)$ denotes Euler's gamma function.

For further and insight information on singularity analysis we refer the reader to [37] and [104].

7. ANALYSIS OF PARTIAL MATCH QUERIES

This chapter covers the average-case analysis of partial match queries in random relaxed K-d trees. This analysis is included in the report *Randomized K-dimensional Search Trees* [28] where the material of [27] comes from.

7.1 The Partial Match Algorithm

Let us start with the description of the partial match algorithm and the previous results given for its average-case performance in standard K-d trees, K-d-t trees and squarish K-d trees.

We briefly review that in a partial match search we are given a query $q = (q_0, q_1, \ldots, q_{K-1})$, with $q_i \in [0,1] \cup \{*\}$ and the goal is to report those points in the file that match the query, that is, the points x such that $x_i = q_i$ if $q_i \neq *$, for all $0 \leq i < K$. For a query q, the bit-string $w = (w_0, w_1, \ldots, w_{K-1})$ such that $w_i = 0$ if $q_i = *$ and $w_i = 1$ otherwise, is called the *specification pattern* of the query. A query might then be thought as a pair consisting in a point $y \in [0,1]^K$ and a bit-string w. Partial matches make sense if at least one coordinate of the query is specified and at least one coordinate is not.

The algorithm for partial match searches over relaxed K-d trees is the same mentioned in Section 2.2 for standard K-d trees. This algorithm explores the tree in the following way. At each node of the K-d tree it examines the corresponding discriminant. If that discriminant is specified in the query then the algorithm recursively follows in the appropriate subtree, depending on the result of the comparison between the attribute of the query and the attribute of the key stored at the current node. Otherwise (that is, the discriminant at the current node is not specified in q), the algorithm recursively follows the two subtrees of the node.

Figure 7.1 shows a relaxed K-d tree together with its induced partition of the search space. Notice that Figure 7.1 shows the same tree that was presented in Figure 3.1, where it was used to show how to build a relaxed K-d tree. Here we will use it to illustrate the behavior of the partial match algorithm over relaxed K-d trees. The dashed line in the partition corresponds to the partial match query $q = (0.550, *)$. The specification pattern of q is then the bit-string $w = 10$. In order to answer query q, the partial match algorithm visits first the root of the tree (x_1). Since the root discriminates by the second attribute and the second attribute is unspecified in query q, the partial match algorithm follows the two subtrees of node x_1. The root of the left subtree (x_2) discriminates also by the second attribute, so the algorithm

visits again the two subtrees of x_2. The root of the left subtree of x_2 has associated the record x_5 and discriminates with respect to the first attribute, which is specified in q. Since the first attribute of q is greater than the first attribute of x_5, the search follows by x_5's right subtree which is empty, and so the search in this branch is finished. The right subtree of x_2, that has associated x_7 to its root, is visited by the algorithm. Since it discriminates with respect to the second attribute, its two empty subtrees are visited and the search by these branches finishes. By following the same procedure, the algorithm proceeds in the left subtree of node x_1, where it visits nodes x_3, x_4 and x_8. The result of the search is that there are no records in the tree that match the query q.

The performance of partial match queries has been extensively studied for several multidimensional data structures (see for instance [17, 24, 38, 59, 72]). The average cost P_n of performing partial match searches in hierarchical K-dimensional data structures of size n, when s out of the K attributes of the query are specified is of the form $P_n = \beta \cdot n^{\alpha(s/K)}$. In the particular case of standard K-d trees, it has been shown by Flajolet and Puech [38] that partial match queries are efficiently supported by random standard K-d trees with an expected cost of $\beta \cdot n^{1-\frac{s}{K}+\theta(\frac{s}{K})}$, where n is the size of the tree and $\theta(x)$ is the unique real solution of

$$(\theta(x) + 3 - x)^x(\theta(x) + 2 - x)^{1-x} - 2 = 0.$$

whose value never exceeds 0.07. No closed expression for the β's is obtained in the paper, but their numerical values are given for $K \leq 4$. The complete characterization of the leading constant β is given by Chern and Hwang [17] using an asymptotic theory for Cauchy-Euler differential equations [16].

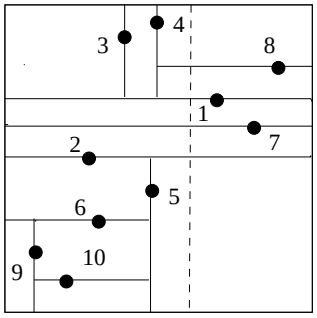
K-d-t trees are similar to standard K-d trees (when $t = 0$ they coincide) but subject to local re-balancing of subtrees of size $\geq 2t + 1$ [22]; for this variant $\theta(x) = \theta_t(x)$ is the unique solution of

$$[(\theta(x) + 3 + t - x)(\theta(x) \cdots (\theta(x) + 3 + 2t - x)]^x \cdot$$
$$[(\theta(x) + 2 + t - x)(\theta(x) \cdots (\theta(x) + 2 + 2t - x)]^{1-x} - \frac{2t + 2!}{t + 1!} = 0.$$

The authors provide the value of β for some specific patterns of the query, as well as the expected cost of standard partial matches for several values of t and n. Once again, characterizations for β are provided by Chern and Hwang [17].

$x_1 = (0.692, 0.703)$
$x_2 = (0.286, 0.495)$
$x_3 = (0.410, 0.895)$
$x_4 = (0.522, 0.953)$
$x_5 = (0.507, 0.394)$
$x_6 = (0.295, 0.300)$
$x_7 = (0.811, 0.605)$
$x_8 = (0.912, 0.807)$
$x_9 = (0.093, 0.210)$
$x_{10} = (0.188, 0.109)$

$q = (0.550, *)$



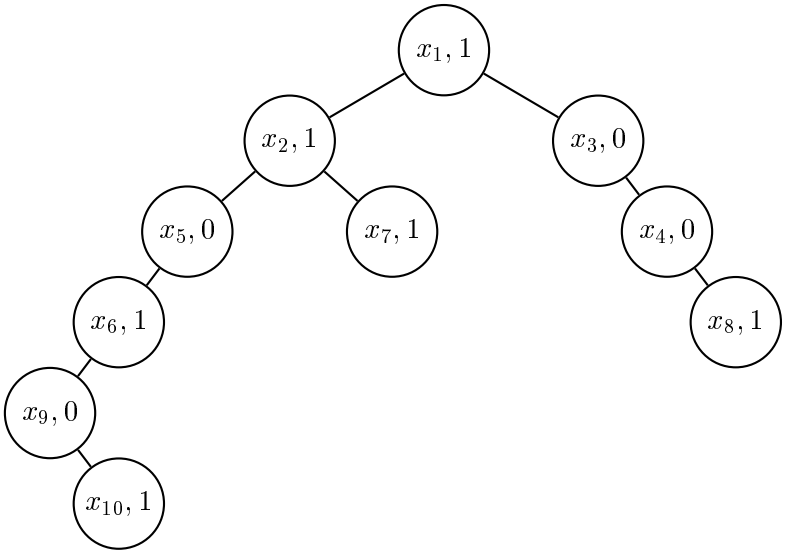


Fig. 7.1: A relaxed 2-d tree, its corresponding induced partition of $[0,1]^2$ and the partial match query q (graphically represented by the dashed line in the partition).

Squarish K-d trees [24] have optimal performance since $\theta(x) = 0$; however, the values of the β's are not yet known and the only way for the moment to compute them would be through experimental measurement.

For quad trees the analysis of partial match can be found in [35]. In this case $\theta(x)$ is the same as for standard K-d trees. But the β's depend only on K and s. For $K = 2$, $\beta = \Gamma(2\alpha + 2)/(2\alpha^3\Gamma^3(\alpha)) \approx 1.5950991$. For higher dimensions, the complete characterization is given by Chern and Hwang [18].

For relaxed K-d tries [72] $\alpha = \alpha(x) = \log_2(2-x)$ and $\beta_w = \beta(\text{order}(w)/K)$ with

$$\beta(x) = \frac{1}{\log 2}\Big((\alpha(x) - 1)\Gamma(-\alpha(x)) + \delta(\log_2 n)\Big),$$

and $\delta(\cdot)$ a periodic function of period 1, mean 0 and small amplitude that also depends on α.

In what follows, we analyze the average cost of performing partial match queries in relaxed K-d trees.

7.2 The Cost of Partial Match Searches

The average-case analysis of the partial match algorithm for relaxed K-d trees is based on the assumption that the K-d tree is random as well as are the queries. The definition of random relaxed K-d trees appears in Section 3.1. For the queries, we say that a partial match query with s out of K attributes specified is random if each attribute has a probability s/K of being specified.

These definitions imply that the partial match algorithm in random relaxed K-d trees with random queries explores the tree in the following way. At each node the corresponding discriminant will be specified in the query with probability $\frac{s}{K}$, then the algorithm will recursively follow the appropriate subtree, depending on the result of the comparison between attributes. With complementary probability (that is $\frac{K-s}{K}$), the corresponding discriminant will be unspecified in the query, so the algorithm will follow the two subtrees recursively.

The following theorem gives the expected performance of a partial match query in a random relaxed K-d tree.

Theorem 7.2.1. *The expected cost P_n (measured as the number of comparisons) of a partial match query with s out of K attributes specified in a random relaxed K-d tree of size n is*

$$P_n = \beta n^{\alpha} + \mathcal{O}(1),$$

where

$$\begin{aligned}\alpha &= \alpha(s/K) = 1 - \frac{s}{K} + \phi(s/K)\\ &\sim_{(s/K)\to 0} 1 - \frac{2}{3}\left(\frac{s}{K}\right) + \mathcal{O}\left(\left(\frac{s}{K}\right)^2\right),\\ \beta &= \frac{\Gamma(2\alpha+1)}{(1-s/K)(\alpha+1)\Gamma^3(\alpha+1)}\end{aligned}$$

with $\phi(x) = \sqrt{9-8x}/2 + x - 3/2$ *and* $\Gamma(x)$ *the Euler's Gamma function* [44].

Proof. Let T be a random relaxed K-d tree of size n with left subtree L and right subtree R, and let $P(T)$ be the average search cost of a partial match search in T. Then, with probability $\frac{K-s}{K}$, the discriminant in the root of T is an unspecified attribute of the query. In this case, the search visits the root and then continues in both subtrees L and R. The cost will be the sum of the cost of visiting the root (one comparison) plus the cost of visiting subtrees L and R which corresponds to $P(L) + P(R)$ and it is reflected by the first term of the right hand side of the equation here-below.
With probability $\frac{s}{K}$ the discriminant in the root of T corresponds to some specified attribute then, after visiting the root, the partial match retrieval continues into the appropriate subtree. It will continue along L with probability $\frac{l+1}{n+1}$ (where l is the size of L) and along R with the complementary probability. In this case, the cost is reflected in the second term of the right hand side of the equation hereafter and it corresponds to the cost of visiting the root plus the cost of visiting either L or R with their corresponding probabilities. Thus, the search cost satisfies the relation

$$\begin{aligned}P(T \mid \mid L \mid = l) &= \frac{K-s}{K}\left(1 + P(L) + P(R)\right)\\ &\quad + \frac{s}{K}\left(1 + \frac{l+1}{n+1}P(L) + \frac{n-l}{n+1}P(R)\right).\end{aligned}$$

Taking the average over all the possible values of l, and since the probability that L has size l is $1/n$ for all l, with $0 \le l < n$, because T is assumed to be random, we find that, for $n > 0$, the expected cost P_n of a partial match query in a random relaxed K-d tree of size n is

$$P_n = 1 + \frac{K-s}{K}\left(\frac{1}{n}\sum_{l=0}^{n-1}[P_l + P_{n-1-l}]\right) + \frac{s}{K}\left(\frac{1}{n}\sum_{l=0}^{n-1}\left[\frac{l+1}{n+1}P_l + \frac{n-l}{n+1}P_{n-1-l}\right]\right),$$

which by symmetry is equivalent to

$$P_n = 1 + 2\left(\frac{K-s}{K}\right)\frac{1}{n}\sum_{l=0}^{n-1} P_l + 2\left(\frac{s}{K}\right)\frac{1}{n}\sum_{l=0}^{n-1}\frac{l+1}{n+1}P_l. \tag{7.2.1}$$

To derive the expression for P_n we use generating functions and singularity analysis [37]. By Definition 6.1.1 the ordinary generating function of the sequence $\{P_n\}_{n\geq 0}$, is $P(z) = \sum_{n\geq 0} P_n z^n$, with $P(0) = 0$. Let us define the generating function $R(z) = \sum_{n\geq 0}(n+1)P_n z^n = zP'(z) + P(z)$, where $R(0) = 0$. Multiplying Equation (7.2.1) by $(n+1)$ gives

$$(n+1)P_n = (n+1) + 2\left(\frac{K-s}{K}\right)\frac{n+1}{n}\sum_{l=0}^{n-1} P_l + 2\left(\frac{s}{K}\right)\frac{1}{n}\sum_{l=0}^{n-1}(l+1)P_l.$$

This recurrence translates to the following integral equation

$$R(z) = \frac{1}{(1-z)^2} - 1 + 2\frac{K-s}{K}\left(\frac{P(z)}{1-z} - P(z) + \int_0^z \frac{P(t)}{1-t}dt\right) + 2\frac{s}{K}\int_0^z \frac{R(t)}{1-t}dt.$$

Taking derivatives and expressing $R(z)$ in terms of $P(z)$ gives the second order non-homogeneous differential equation

$$P''(z) - 2\frac{(2z-1)P'(z)}{z(1-z)} - 2\frac{(2-s/K-z)P(z)}{z(1-z)^2} - 2\frac{1}{z(1-z)^3} = 0. \tag{7.2.2}$$

The homogeneous differential equation associated to Equation (7.2.2) has only z and $(1-z)^p$ as divisors for $p = 1, 2$. Thus, $P(z)$ has a single singularity at $z = 1$ and because p is integer, the function is meromorphic with a single pole at $z = 1$. Thus, the dominant contribution in the local expansion of $P(z)$, when $z \to 1$, is of the form $P(z) \sim \beta(1-z)^{\xi}$, with ξ the smallest root of the indicial equation: $\xi^2 + \xi - 2(1 - s/K) = 0$, which results in $\xi = \frac{-1-\sqrt{(9-8s/K)}}{2}$.
Because of Lemma 6.2.2 it follows that,

$$P_n = [z^n]P(z) \sim \beta n^{\alpha} \tag{7.2.3}$$

for $\alpha = -\xi - 1$ and some constant β.
Expanding $\alpha(s/K)$ in Taylor series, we obtain that $\alpha(s/K) \sim 1 - 2/3(s/K) + \mathcal{O}((s/K)^2)$.

The approach that we have sketched above for the analysis of partial match can be further explored finding the result,

$$P_n = [z^n]P(z) = \beta n^\alpha + \mathcal{O}(1), \tag{7.2.4}$$

which is stronger than (7.2.3). In fact in [72] it has been shown that the exact solution of Equation (7.2.2) is,

$$P(z) = \frac{1}{1-s/K}\left({}_2F_1\left(\begin{matrix}a,b\\2\end{matrix}\middle| z\right)(1-z)^\xi - \frac{1}{1-z}\right), \tag{7.2.5}$$

where ${}_2F_1\left(\begin{smallmatrix}a,b\\c\end{smallmatrix}\middle| x\right)$ is the hypergeometric function [44], with $a = 2+\xi$ and $b = 1+\xi$. Then, we can study the asymptotic behavior of $P(z)$ when $z \to 1$ to get not only the precise order of magnitude of P_n but the coefficient of the main order term and the magnitude of the lower order terms. The second term of $P(z)$ makes a contribution which is $\mathcal{O}(1)$ and the hypergeometric function is analytic at $z = 1$. Therefore,

$$\beta = \frac{{}_2F_1\left(\begin{smallmatrix}a,b\\2\end{smallmatrix}\middle| 1\right)}{(1-s/K)\Gamma(-\xi)} = \frac{\Gamma(2\alpha+1)}{(1-s/K)(\alpha+1)\Gamma^3(\alpha+1)}.$$

□

It is interesting to point out that, although Theorem 7.2.1 is valid only if $0 < \frac{s}{K} < 1$, it provides meaningful information for the limiting cases-at least, to some extent. If $\frac{s}{K} \to 0$, that is, no attribute is specified, then $P_n = n$. Indeed, $\alpha \to 1$ and $\beta \to 1$ as $\frac{s}{K} \to 0$. On the other hand, in an exact match all attributes are specified and $s = K$. In this case, we know that $P_n = \Theta(\log n)$. And we have that $\alpha \to 0$ and $\beta \to \infty$ if $\frac{s}{K} \to 1$, which is an approximate way to say with a formula like βn^α that P_n grows with n, but slower than any function of the type n^ϵ, for real positive ϵ.

In Figures 7.2 and 7.3 we plot respectively the value of the exponent α in the average cost of partial match queries in random K-d trees and random relaxed K-d trees, and the excess of the corresponding exponents with respect to $1 - \frac{s}{K}$, since $\Theta(n^{1-s/K})$ is the best known lower bound for partial match queries. In figure 7.4 we plot the value of β as a function of the ratio $\rho = s/K$.

Observe that the expected cost of partial match queries in random relaxed K-d trees is slightly higher than the one given by Flajolet and Puech [38]

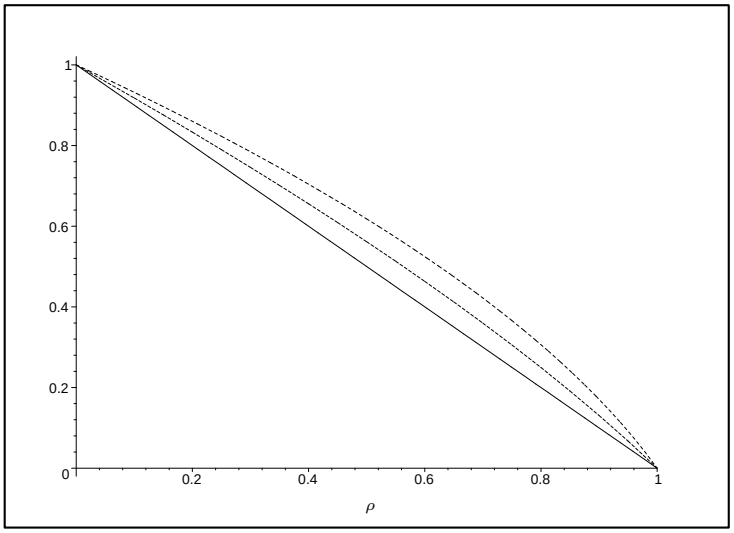


Fig. 7.2: The value of the exponent in the average cost of partial match queries in standard (middle dotted line) and relaxed (upper dashed line) random K-d trees. The solid line corresponds to the lower bound $n^{1-s/K}$.

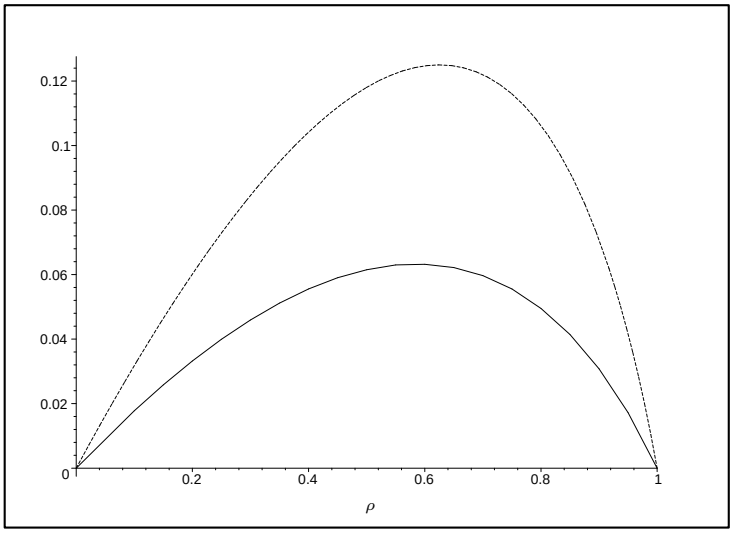


Fig. 7.3: Excess (with respect to $1-\rho$) of the exponent in the average cost of partial match queries in relaxed (dashed line) and standard (solid line) random K-d trees.

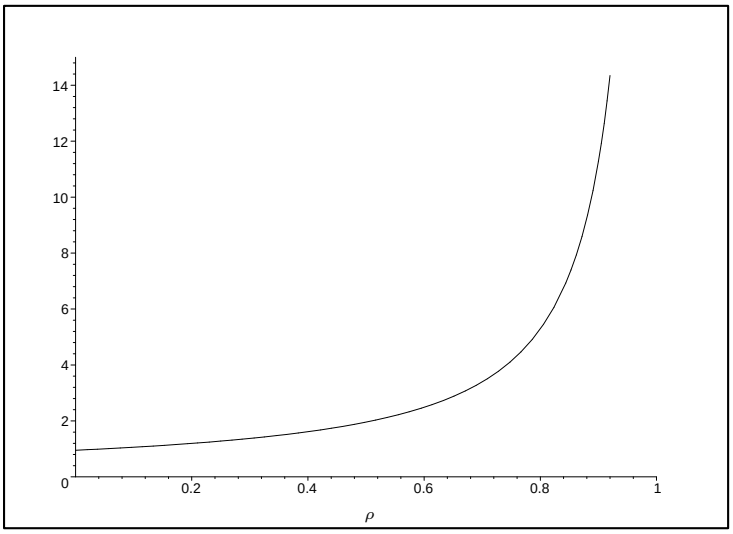


Fig. 7.4: The value of the constant β in the average cost of partial match in random relaxed K-d trees.

for random K-d trees. In fact, the values of $\phi(s/K)$ never exceed 0.12. It is possible to show that the difference in the exponent of n of these costs is at most 0.08, though-and for extreme values of s/K the difference is much smaller. Notice also that the constant β in the main order term of the expected cost of partial match queries in relaxed K-d trees is independent of the specification pattern of the query whereas for standard K-d trees such a constant is dependent on the particular pattern of the query [17, 38].

8. ANALYSIS OF ORTHOGONAL RANGE QUERIES

Orthogonal range queries are in the basis of associative retrieval because apart from the applications of range search as such, they are involved in more complex region queries and other associative queries.

In this chapter we give exact upper and lower bounds (Theorems 8.1.2 and 8.1.3) and a characterization of the cost of range search as the sum of the cost of partial match-like searches (Theorem 8.1.4). We then use these results to obtain asymptotic estimates for the expected cost of range search (Theorem 8.1.5). Our proof techniques—a combination of geometric and combinatorial arguments— are rather different from those in [14, 24] and they are easily applicable to many multidimensional data structures.

We analyze first the average cost of range search in randomized K-d trees [27]. We begin introducing, in Section 8.1, *sliced partial matches* and then we relate the performance of range search with the performance of sliced partial matches; we use this relationship to provide a tight asymptotic analysis of the average cost of range search. In Section 8.2 we show that the results for randomized K-d trees can be easily extended to most tree-like multidimensional data structures, namely, standard K-d trees, squarish K-d trees, K-d-t trees, standard and relaxed K-d tries, quad trees and quad tries. The cost of nearest neighbor search in relaxed K-d trees is mentioned in Section 8.3. In the last section, we report the results of an experimental study conducted in order to validate the obtained analytic results. The results of this chapter are included in the papers titled *On the Average Performance of Orthogonal Range Search in Multidimensional Data Structures* [29, 30].

8.1 The Cost of Range Searches

Our average-case analysis of range searches over relaxed K-d trees will assume that trees are *random* (see Definition 3.1.2). We will also assume random range queries. We will use, in this case, a small variation of the probabilistic model of random range queries introduced in [14, 24]. In our model, the edges of a *random range query* have given lengths $\Delta_0, \Delta_1, \ldots \Delta_{K-1}$, with $0 \leq \Delta_i \leq 1/2$, for $0 \leq i < K$ and the center of the query is an independently drawn point z in

$$\begin{aligned} Z_\Delta = &\prod_{0\leq r<K} [-\Delta_r/2, 1+\Delta_r/2] \\ =&[-\Delta_0/2, 1+\Delta_0/2] \times [-\Delta_1/2, 1+\Delta_1/2] \times \cdots \times [-\Delta_{K-1}/2, 1+\Delta_{K-1}/2], \end{aligned}$$

sampled from some continuous distribution. Therefore, $\ell_i = z_i - \Delta_i/2$ and $u_i = z_i + \Delta_i/2$, for $0 \leq i < K$. Notice that in this model a range query Q may fall partially outside of $[0,1]^K$, so in general,

$$Q \subset C_\Delta = \prod_{0 \leq r < K} [-\Delta_r, 1 + \Delta_r].$$

Range searching in any variant of K-d trees is straightforward. We refer the reader to Algorithm 1 in Section 2.2. This algorithm works very similar to the partial match algorithm described in the previous chapter.

We will measure the cost of range queries by the number of nodes of the K-d tree visited during the search. If the number of points to be reported by the range search is P then the cost R_n of the range search will be of the form $\Theta(P + W_n)$, where W_n is the *overhead*.

8.1.1 Slices and Sliced Partial Match

In order to relate the performance of range searches with the performance of partial matches, we need to introduce several notions. We begin with that of *slice*. Given a bit-string $w = (w_0, \ldots, w_{K-1})$ of length K, the slice Q_w is the K-dimensional hyper-rectangle defined by

$$Q_w = \prod_{0 \leq r < K} [\ell'_r, u'_r],$$

where $[\ell'_i, u'_i] = [\max\{0, \ell_i\}, \min\{u_i, 1\}]$ if $w_i = 0$ and $[\ell'_i, u'_i] = [0,1]$ if $w_i = 1$. Notice that $Q_{00\ldots0} = Q \cap [0,1]^K$ and $Q_{11\ldots1} = [0,1]^K$.

Another useful notion is that of *proper slice*. The proper slice $\hat{Q}_w$ is the hyper-region defined by

$$\hat{Q}_w = Q_w - \bigcup_{v < w} Q_v,$$

where $v < w$ if and only if $v_i < w_i$ for all $0 \leq i < K$. Thus a proper slice $\hat{Q}_w$ is the region that results when all properly contained slices within Q_w are subtracted from it (see Fig. 8.1). Alternatively, $\hat{Q}_w$ is the result of subtracting from Q_w those slices Q_v such that $v < w$ and v differs from w in just one bit. The unique proper slice consisting of a simple connected region is $\hat{Q}_{00\ldots0} = Q_{00\ldots0} = Q \cap [0,1]^K$; in general, $\hat{Q}_w$ consists of $2^{\text{order}(w)}$ connected subregions, where order(w) is the number of 1's in the bit-string w.

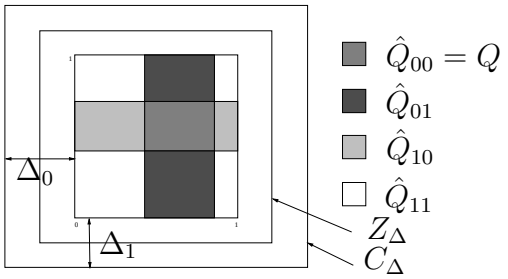


Fig. 8.1: Example of the proper slices induced by a query Q

The most important concept in this subsection is that of *sliced partial match*.

Given a query hyper-rectangle Q, a bit-string w and a point $y \in C_\Delta$, a sliced partial match acts as a standard partial match with query $q = (q_0, q_1, \ldots, q_{K-1})$ where $q_i = y_i$ if $w_i = 1$ and $q_i = *$ if $w_i = 0$ (hence the specification pattern of the partial match is w), but contrary to a standard partial match it only reports the visited points x in the data structure such that $x \in \hat{Q}_w$.

To every sliced partial match with point y and specification pattern w we associate the hyperplane $H(y, w)$ defined by

$$H(y, w) = \{x \in C_\Delta \,|\, \forall i : w_i = 1 \implies x_i = y_i\}.$$

Notice that the value of y_i in the definition of $H(y, w)$ is irrelevant if $w_i = 0$.

In the development of the results included in this chapter we will extensively use Lemma 5.1.1 that relates the notion of bounding hyper-rectangles (given in Definition 2.2.3) with the range search algorithm.

Lemma 8.1.1. *A point x with bounding hyper-rectangle $B(x)$ is visited and reported by a partial match with query point y and specification pattern w, if and only if the bounding rectangle $B(x)$ intersects the hyperplane $H(y, w)$.*

A point x with bounding rectangle $B(x)$ is visited and reported by a sliced partial match with query hyper-rectangle Q, specification pattern w, and query point y, if and only if $x \in \hat{Q}_w$ and the bounding rectangle $B(x)$ intersects the hyperplane $H(y, w)$.

Proof. The proof is immediate from Lemma 5.1.1. Notice that a (sliced) partial match behaves as a range query in which the hyper-rectangle query

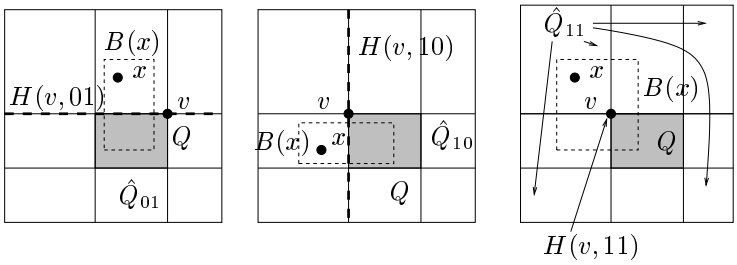


Fig. 8.2: Graphical illustration of the proof of Theorem 8.1.2

"degenerates" to the hyperplane $H(y, w)$ (when a coordinate is specified in the query the corresponding range $[\ell_i, u_i]$ has identical endpoints; when the coordinate is not specified we have a corresponding full range). In the case of sliced partial matches, only those points that also belong to $\hat{Q}_w$ are reported. □

8.1.2 The Combinatorial Characterizations

In this subsection we state several relations between the cost $R(t)$ of an orthogonal range search in a K-d tree t and the performance $P_w(t, y)$ of a sliced partial match with specification pattern w and query point y in a K-d tree t. The implicit query hyper-rectangle Q is the same for both the range search and the sliced partial match.

Theorem 8.1.2. *Given a query Q with corners $v_0, v_1, \ldots, v_{2^K-1}$ and a K-d tree t,*

$$R(t) \leq \sum_{0 \leq j < 2^K} \sum_{w \in (0+1)^K} P_w(t, v_j).$$

Proof. Consider a point x visited by a range search with query Q. Let w be the index of the proper slice that contains x, i.e., $x \in \hat{Q}_w$. Recall that since x is visited by the range search we have $B(x) \cap Q \neq \emptyset$ (Lemma 5.1.1). Therefore, by Lemma 8.1.1, it suffices to show that if $B(x)$ intersects Q then there exists at least one corner v_j of Q such that the hyperplane $H(v_j, w)$ does intersect $B(x)$.
If $B(x)$ contains any of the corners of Q then the statement above is clearly true: since the hyperplane $H(v_j, w)$ contains v_j, it must intersect $B(x)$. If

$B(x)$ does not contain any corner of Q, there are two possibilities to consider: either $B(x)$ is entirely within Q, or $B(x)$ intersects one or more faces of Q. If $B(x)$ is totally inside Q then $w = 0 \ldots 0$ and indeed $H(v_j, 0 \ldots 0) = C_\Delta$ intersects $B(x)$ for any corner v_j. On the other hand, if $B(x)$ intersects one or more faces of Q but does not contain a corner nor it is contained inside Q then $w \neq 11 \ldots 1$ and $\hat{Q}_w$ must "contact" one of the intersected faces, in the sense that the face is a boundary of $\hat{Q}_w$. Let f be such face. Now, the hyperplane $H(v_j, w)$ contains this face (and hence it intersects $B(x)$), provided that v_j is any corner of the face f (see Fig. 8.2 for a graphical illustration of this proof when $K = 2$). □

Theorem 8.1.3. *Given a query Q with center at z and a K-d tree t,*

$$R(t) \geq \sum_{w \in (0+1)^K} P_w(t, z).$$

Proof. The statement of the theorem is immediate, once we show that whenever a point x is reported by a sliced partial match with parameters w and z then it is also visited by the range search with query Q. Formally, we have to show that if $x \in \hat{Q}_w$ and $H(z, w)$ intersects the bounding rectangle $B(x)$ of x then $B(x)$ also intersects Q. Indeed, if $w = 00 \ldots 0$ then the statement trivially holds since $x \in Q$. On the other hand, if $w \neq 00 \ldots 0$ then $H(z, w)$ and $\hat{Q}_w$ are disjoint. Since $B(x)$ intersects $\hat{Q}_w$ (x is part of both by hypothesis) the only way for $B(x)$ to intersect $H(z, w)$ is to intersect Q too (then, because of Lemma 5.1.1, x must be visited by the range search). □

Theorem 8.1.4. *Given a query Q with center at z divide C_Δ into 2^K quadrants $C_0, C_1, \ldots, C_{2^K-1}$, with z the contact point of the 2^K quadrants. Let $R^{(i)}(t)$ denote the number of points of the i–th quadrant visited by a range search with query Q in the K-d tree t. Similarly, let $P_w^{(i)}(t, y)$ denote the number of points of the i–th quadrant reported by sliced partial match with pattern w and point y in the K-d tree t. Let v_i be the unique corner of Q belonging to the i–th quadrant. Then*

$$R^{(i)}(t) = \sum_{w \in (0+1)^K} P_w^{(i)}(t, v_i).$$

Proof. For a point x belonging to the i-th quadrant and the proper slice $\hat{Q}_w$, the intersection of $H(v_i, w)$ with $B(x)$ implies the intersection of Q with $B(x)$. On the other hand, if $B(x)$ intersects Q there is at least one corner v such that $H(v, w)$ intersects $B(x)$; it is not difficult to see that one of these corners must be v_i, the unique corner of Q in the i-th quadrant (Fig. 8.2 may also help understanding the proof even though quadrants are not depicted there). □

8.1.3 The Cost of Range Search in Relaxed K-d Trees

The theorems of Subsection 8.1.2 show that the analysis of range search reduces to the analysis of sliced partial matches.

Theorem 8.1.5. *Let $\mathbb{E}[R_n]$ be the expected cost of a range search with a random query in a random K-d tree of size n. Let $\mathbb{E}[P_{n,w}]$ be the expected cost of a sliced partial match with pattern w in a random K-d tree of size n, with respect to a uniformly and independently drawn random query point in $[0,1]^K$. Then*

$$\frac{1}{V(Z_\Delta)} \cdot \sum_{w \in (0+1)^K} \mathbb{E}[P_{n,w}] \leq \mathbb{E}[R_n] \leq V(Z_\Delta) \cdot \sum_{w \in (0+1)^K} \mathbb{E}[P_{n,w}],$$

where $V(Z_\Delta) = \prod_{0 \leq r < K}(1 + \Delta_r)$.

Proof. Clearly $R(t) = \sum_{0 \leq i < 2^K} R^{(i)}(t)$. Hence,

$$R(t) = \sum_{0 \leq i < 2^K} \sum_{w \in (0+1)^K} P_w^{(i)}(t, v_i). \tag{8.1.1}$$

Given a corner v of a query Q, if $v \in [0,1]^K$ let $v' = v$, otherwise let v' be the point in the boundary of $[0,1]^K$ closest to v (see Fig. 8.3). It is pretty clear that if Q falls partially off the $[0,1]^K$ boundary, the cost of the range search is the same as if we had a range query Q' where we had chopped the part of Q that falls outside $[0,1]^K$. And if we shift a query so that a corner v outside $[0,1]^K$ is aligned to v' then the corresponding range search will have a cost which is greater or equal to the cost of a range search where we do not perform such a shift. In other words, for any bit-string w, query Q, K-d tree t and quadrant i,

$$P_w(t, v_i) \leq P_w(t, v_i').$$

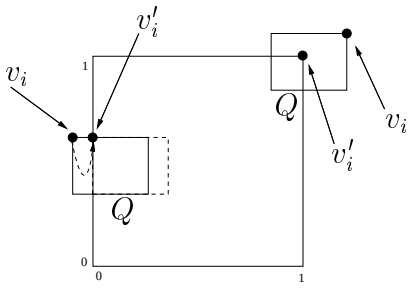


Fig. 8.3: Shifting queries falling partially off the boundaries.

Hence,

$$R(t) \leq \sum_{0 \leq i < 2^K} \sum_{w \in (0+1)^K} P_w^{(i)}(t, v_i').$$

The next step is to take expectations on both sides of the equation above and observe that, for uniformly distributed centers in Z_Δ, the probability that v_i falls outside $[0,1]^K$ is

$$\left(\prod_{0 \leq r < K} (1 + \Delta_r) \right) - 1.$$

Then we have

$$\mathbb{E}\left[R_n\right] \leq \prod_{0 \leq r < K} (1 + \Delta_r) \cdot \sum_{w \in (0+1)^K} \mathbb{E}\left[P_{n,w}^{(i)}\right],$$

where $\mathbb{E}\left[P_{n,w}^{(i)}\right]$ is the expected cost of a sliced partial match with respect to the i-th quadrant and a random uniformly distributed query point in $[0,1]^K$. Now,

$$\mathbb{E}\left[P_{n,w}^{(i)}\right] = \mathrm{Vol}(C_i) \cdot \mathbb{E}\left[P_{n,w}\right],$$

where $\mathrm{Vol}(C_i)$ is the probability that, given a randomly drawn point z in Z_Δ, a random data point in $[0,1]^K$ falls in the i-th quadrant defined by z. Since $\sum_{0 \leq i < 2^K} \mathrm{Vol}(C_i) = 1$ the upper bound follows.
For the lower bound we use Theorem 8.1.3. Taking expectations in both sides of the inequality of Theorem 8.1.3 and conditioning on the event "the center of the query is inside $[0,1]^K$" the lower bound given in the statement is immediate, as the probability that the center of a random query falls inside $[0,1]^K$ is $1/\prod_{0 \leq r < K}(1 + \Delta_r)$. □

Although (8.1.1) gives a precise relationship between the cost of range search and the cost of sliced partial matches, we cannot use it to get results about the variance of R_n or its probability distribution since the costs of the sliced partial matches (the random variables $P_{n,w}^{(i)}$) are not independent.

Now we need to analyze the expected cost of sliced partial matches in random(ized) relaxed K-d trees. It easily follows from the analysis of the expected cost of standard partial matches in random relaxed K-d trees [72]. Our next theorem gives the expected cost of sliced partial matches in random relaxed K-d trees.

Theorem 8.1.6. *If $w \neq 00\ldots0$ and $w \neq 11\ldots1$, the expected cost $\mathbb{E}[P_{n,w}]$ of a sliced partial match in a random relaxed K-d tree of size n w.r.t. a random query point in $[0,1]^K$ and the pattern w is*

$$\mathbb{E}[P_{n,w}] = \mathit{Vol}(\hat{Q}_w) \cdot \beta(\rho) \cdot n^{\alpha(\rho)} + \mathcal{O}(1),$$

where $\rho = \mathit{order}(w)/K$, $\alpha \equiv \alpha(x) = (\sqrt{9-8x}-1)/2$, $\beta(x) = \Gamma(2\alpha+1)/((1-x)(\alpha+1)\alpha^3\Gamma^3(\alpha))$, and $\mathit{Vol}(\hat{Q}_w)$ is the probability that a data point falls inside the proper slice $\hat{Q}_w$ of a randomly centered query. Furthermore, $\mathbb{E}[P_{n,00\ldots0}] = \mathit{Vol}(\hat{Q}_{00\ldots0}) \cdot n$ and $\mathbb{E}[P_{n,11\ldots1}] = 2 \cdot \mathit{Vol}(\hat{Q}_{11\ldots1}) \cdot (H_{n+1}-1)$, where $H_n = \sum_{1\leq j\leq n} 1/j = \log n + \gamma + \mathcal{O}(1/n)$ denotes the n-th harmonic number.

Proof. The recurrence for $\mathbb{E}[P_{n,w}]$ is

$$\mathbb{E}[P_{n,w}] = \mathrm{Vol}(\hat{Q}_w) + \frac{1}{n}\sum_{0\leq k<n}\Big[\rho\cdot\left(\frac{k+1}{n+1}\mathbb{E}[P_{k,w}] + \frac{n-k}{n+1}\mathbb{E}[P_{n-k,w}]\right) + (1-\rho)\cdot(\mathbb{E}[P_{k,w}] + \mathbb{E}[P_{n-k,w}])\Big],$$

where ρ and $\mathrm{Vol}(\hat{Q}_w)$ are defined as in the statement of the theorem.
Let $y_w(x) = \sum_{n\geq 0}\mathbb{E}[P_{n,w}]\, x^n$. Then it is easy to show that $y_w(x)$ satisfies

$$xy_w''(x) - 2\frac{2x-1}{1-x}y_w'(x) - 2\frac{2-\rho-x}{(1-x)^2}y_w(x) - 2\frac{\mathrm{Vol}(\hat{Q}_w)}{(1-x)^3} = 0, \qquad (8.1.2)$$

and the initial conditions are $y_w(0) = 0$ and $y_w'(0) = \mathrm{Vol}(\hat{Q}_w)$. The linear differential equation satisfied by the generating function $y(x)$ of the expected cost of standard partial matches is almost the same as (8.1.2), except that the independent term there is $-2/(1-x)^3$ and $y'(0) = 1$. It is straightforward

then to show that $y_w(x) = \mathrm{Vol}(\hat{Q}_w) \cdot y(x)$. The statement of the theorem now immediately follows from the known asymptotic estimates for the expected cost of standard partial matches in random relaxed K-d trees [72]. The special cases $w = 00\ldots0$ ($\rho = 0$) and $w = 11\ldots1$ ($\rho = 1$) are similarly handled; they are even easier, as the first behaves like a full traversal of the tree, whereas the second behaves like an exact search in a binary search tree. □

Computing the volumes $\mathrm{Vol}(\hat{Q}_w)$ of proper slices turns out to be a difficult task, both because our model allows queries to fall partially off the boundaries and the data points do not have to be uniformly distributed. On the other hand, if the Δ_i's are large then the gap between the lower and upper bounds of Theorem 8.1.5 is significant; besides, the random model loses interest as the "frame" around the data region is also too large. But if the Δ_i's tend to 0 as $n \to \infty$ (in other words, the number of reported points does not grow linearly with n) and the data points are uniformly distributed, then we can easily establish the following corollary.

Corollary 8.1.7. *Given a random relaxed K-d tree storing n uniformly and independently drawn data points in $[0,1]^K$, the expected cost of a random range search of sides $\Delta_0, \ldots, \Delta_{K-1}$ (with $\Delta_i \to 0$ as $n \to \infty$) with center uniformly and independently drawn from Z_Δ, is given by*

$$\mathbb{E}[R_n] \sim \Delta_0 \cdots \Delta_{K-1} \cdot n + \sum_{1 \le j < K} c_j \cdot n^{\alpha(j/K)} + 2 \cdot (1-\Delta_0) \cdots (1-\Delta_{K-1}) \cdot \log n + \mathcal{O}(1),$$

where

$$c_j = \beta(j/K) \cdot \sum_{w : order(w) = j} \left(\prod_{i : w_i = 0} \Delta_i \right) \cdot \left(\prod_{i : w_i = 1} (1 - \Delta_i) \right).$$

Proof. The corollary follows from Theorem 8.1.5 and the asymptotic estimates given in Theorem 8.1.6. When $\Delta_i \to 0$ as $n \to \infty$, we have $V(Z_\Delta) \to 1$; hence the lower and upper bounds in Theorem 8.1.5 match and

$$\mathbb{E}[R_n] \sim \sum_{w \in (0+1)^K} \mathbb{E}[P_{n,w}].$$

Finally, observe that for the uniform distribution, and provided that the Δ_i's are small enough, we have

$$\mathrm{Vol}(\hat{Q}_w) \sim \left(\prod_{i:w_i=0} \Delta_i\right) \cdot \left(\prod_{i:w_i=1} (1-\Delta_i)\right).$$

□

Notice that the term $\Delta_0 \cdots \Delta_{K-1} \cdot n$ in $\mathbb{E}[R_n]$ is the expected number of reported points and hence the overhead is $\mathcal{O}(n^{\alpha(1/K)})$.

The result above assumes a random model where the queries may fall partially outside $[0,1]^K$, but it also gives a very good approximation to the average cost of range searches in a random model where the queries must completely fall inside $[0,1]^K$ (again, provided that $\Delta_i \to 0$ for $0 \le i < K$ and the centers and data points are uniformly distributed).

8.2 Other Multidimensional Data Structures

It is important to stress that no assumptions were made with respect to the way that discriminants are assigned during the construction of the tree, so all theorems of subsection 8.1.2 and Theorem 8.1.5 apply to standard K-d trees [4], squarish K-d trees [24], K-d-t trees [22] and other variants. It turns out that the theorems also apply to quad trees [6] without change.

Furthermore, similar arguments to that of Theorem 8.1.6 are also valid, relating the expected cost of sliced partial matches to the expected cost of standard partial matches. In general, when $w \neq 00\ldots0$ and $w \neq 11\ldots1$ for the data structures mentioned above we have

$$\mathbb{E}[P_{n,w}] = \beta_w \cdot \mathrm{Vol}(\hat{Q}_w) \cdot n^{\alpha(\rho)} + \mathcal{O}(1),$$

where $\rho = \mathrm{order}(w)/K$, $\alpha(x) = 1 - x + \phi(x)$ and β_w is a constant depending on w. The expected cost of range search, when $\Delta_i \to 0$, takes hence the form

$$\begin{aligned}\mathbb{E}[R_n] = \Delta_0 \cdots \Delta_{K-1} \cdot n + \sum_{1 \le j < K} c_j \cdot n^{\alpha(j/K)} \\ + 2 \cdot (1-\Delta_0) \cdots (1-\Delta_{K-1}) \cdot \log n + \mathcal{O}(1),\end{aligned} \tag{8.2.1}$$

where $c_j = \sum_{w:\mathrm{order}(w)=j} \beta_w \cdot \mathrm{Vol}(\hat{Q}_w)$,

Different data structures are characterized by different α's and β's.

The theorems in Section 8.1 also apply to K-d tries, relaxed K-d tries and quad tries. In order to apply the theorems of Section 8.1, we only need to assume that each internal node "contains" the middle point of the hyperplane(s) associated to the internal node, to meaningfully define sliced partial matches in these data structures. The average cost of range searches in these digital data structures satisfies (8.2.1), but the β_w's involve a fluctuating periodic term (depending on n) of small amplitude and bounded by a constant.

8.3 A Note on Nearest Neighbor Search

The performance of nearest neighbor search is similar to the overwork in range search. Given a random K-d tree, the expected cost $\mathbb{E}[NN_n]$ of a nearest neighbor query q uniformly drawn from $[0,1]^K$ is given by the following expression [14],

$$\mathbb{E}[NN_n] = \Theta(n^\rho + \log n),$$

where $\rho = \max_{0\le s\le K}(\alpha(s/K) - 1 + s/K)$. In the case of standard K-d trees $\rho \in (0.0615, 0.064)$. More precisely, for $K = 2$ we have $\rho = (\sqrt{17} - 4)/2 \approx 0.0615536$ and for $K = 3$ we have $\rho \approx 0.0615254$, which is minimal.

For relaxed K-d trees $\rho \in (0.118, 0.125)$. When $K = 2$ we have $\rho = \frac{\sqrt{5}}{2} - 1 \approx 0.118$, which is minimal, whereas for $K = 8$ we have $\rho = \frac{1}{8}$, which is maximal.

8.4 Experimental Results

We have conducted a series of experiments to validate the theoretical analysis of the previous sections and to explore its limitations, when the input data does not fulfill some of the hypotheses of the random model.

Each "sample point" in our experiments consisted of a random relaxed (T_r) and standard (T_s) K-d tree of size n and dimension K, both built from the same sequence of random insertions. Up to Q range searches with random queries of given edge lengths were requested in both trees. And this was repeated for T pairs (T_r, T_s) of trees of each size and dimension.

The legends of the plots indicate the different parameter values: dimension (K), size (n), number of trees per size (T), number of queries per tree (Q) and edge lengths (Δ). Each plot depicts the empirical expected cost of range search against the theoretical predicted value.

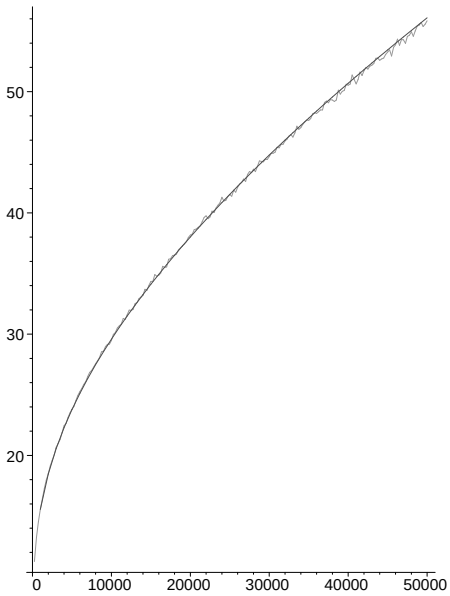


Fig. 8.4: Relaxed K-d trees ($K = 2$, $n \leq 50000$, $T = 300$, $Q = 100$, $\Delta = [0.01, 0.01]$).

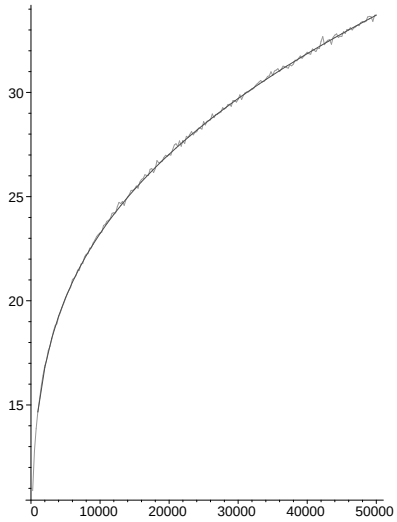


Fig. 8.5: Relaxed K-d trees ($K = 3$, $n \leq 50000$, $T = 300$, $Q = 100$, $\Delta = [0.01, 0.01, 0.01]$).

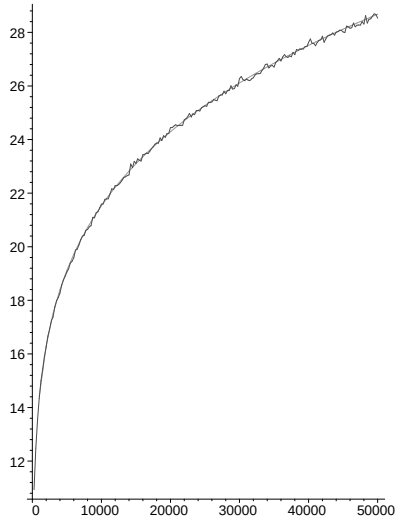


Fig. 8.6: Relaxed K-d trees ($K = 4$, $n \leq 50000$, $T = 300$, $Q = 100$, $\Delta = [0.01, 0.01, 0.01, 0.01]$).

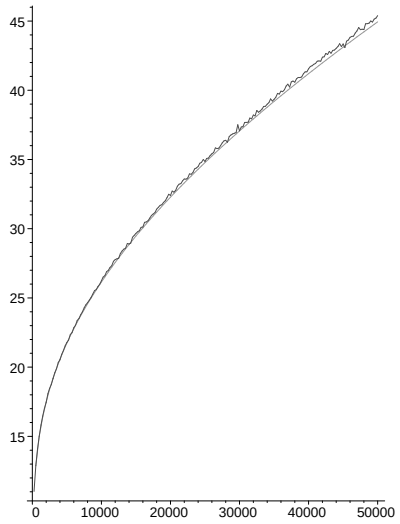


Fig. 8.7: Standard K-d trees ($K = 2$, $n \leq 50000$, $T = 300$, $Q = 100$, $\Delta = [0.01, 0.01]$).

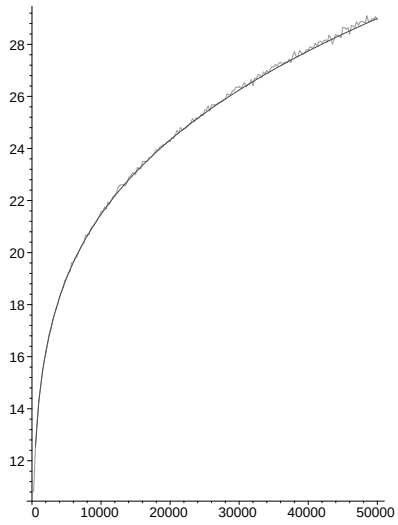


Fig. 8.8: Standard K-d trees ($K = 3$, $n \leq 50000$, $T = 300$, $Q = 100$, $\Delta = [0.01, 0.01, 0.01]$).

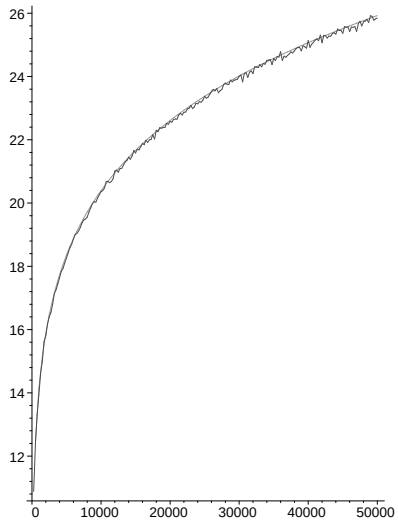


Fig. 8.9: Standard K-d trees ($K = 4$, $n \leq 50000$, $T = 300$, $Q = 100$, $\Delta = [0.01, 0.01, 0.01, 0.01]$).

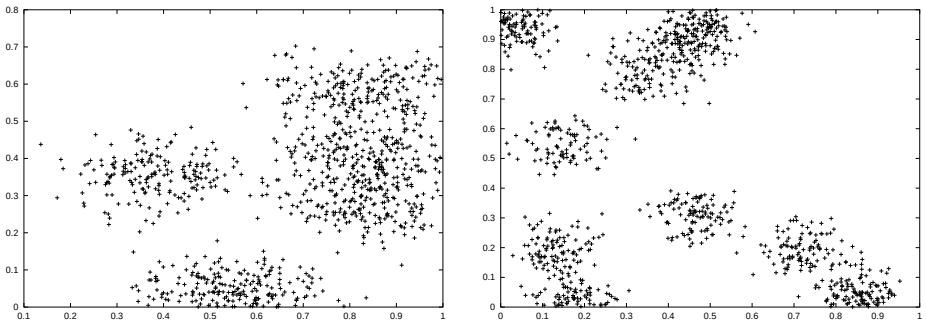


Fig. 8.10: Two clustered distributions of $n = 1000$ points in $[0,1]^2$ (left: $c = 5, \sigma = [0.1, 0.05]$; right: $c = 10, \sigma = [0.05, 0.05]$).

The experiments suggest that the variance of the cost of range search is high. We conjecture that if $\mathbb{E}[R_n] = a \cdot n + b \cdot n^\alpha + o(n^\alpha)$, the variance of R_n is $\Theta(n^{2\alpha})$.

We have also performed experiments using the same setting as for the first set of experiments, but the data points and query centers were drawn from a clustered distribution (see for example Fig. 8.10).

In order to generate these clustered distributions, a number of "clusters" is fixed in advance and the centers of the clusters uniformly and independently generated in $[0,1]^K$. To generate a data point, a cluster is selected at random with identical probability and each coordinate of the point is then generated according to a normal law whose mean is the corresponding coordinate of the center of the cluster. If the generated data point falls outside $[0,1]^K$ then it is rejected and the procedure repeated.

In our experiments, the number of clusters was $c = 10$ and the standard deviations σ_i of the normal laws were taken all identical to 0.05. The centers of the queries were generated according to the same distribution as the data points.

There are two problems for the application of Corollary 8.1.7 to data and range queries generated according to the model above. First, it is very difficult, if not impossible, to analytically compute the volumes of the proper slices. We have prepared a program to compute the approximate values of these volumes for given Δ's, by repeatedly "throwing" queries with the given edge lengths and a large number of data points for each query, keep-

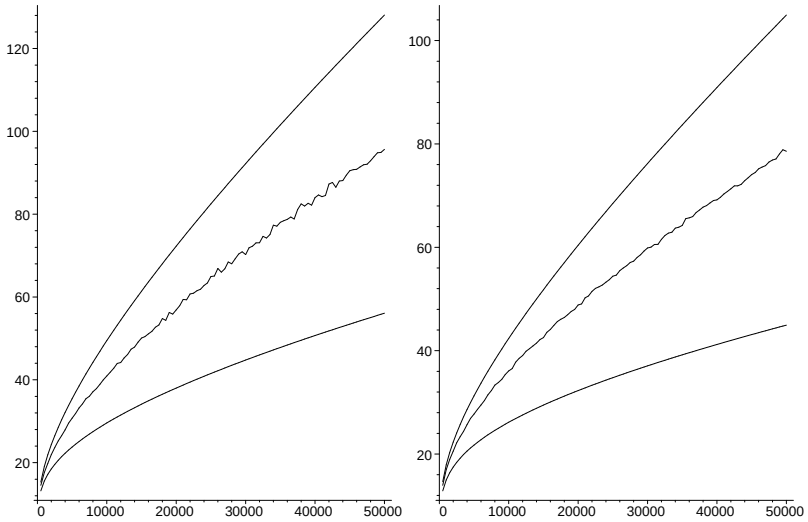


Fig. 8.11: Clustered distribution, $K = 2$, $n \leq 50000$, $T = 50$, $Q = 1000$, $\Delta = [0.01, 0.01]$. Left: Relaxed K-d trees; right: standard K-d tree. The upper curves in the plots are the theoretical predictions with estimated proper slices' volumes; the middle curves show empirically measured costs; the lower curves are the theoretical predictions with uniform estimates for proper slices' volumes.

ing frequency counts of the event "point falls in $\hat{Q}_w$". For instance, when $K = 2$ and $\Delta = [0.01, 0.01]$ we have the estimates $\text{Vol}(\hat{Q}_{00}) = 0.000868$, $\text{Vol}(\hat{Q}_{01}) = 0.022335$, $\text{Vol}(\hat{Q}_{10}) = 0.019432$ and $\text{Vol}(\hat{Q}_{11}) = 0.957365$, which are significantly different from the volumes for the given values of Δ in the uniform distribution.

The second and most important problem is that the data points and query centers are not independent, so that Corollary 8.1.7 cannot actually be applied. However, our experiments show that it still provides reasonable approximations to the observed data.

The plots for clustered input show the results of the experiments together with: a) the estimates given by Corollary 8.1.7 as if the points where uniformly distributed; b) the estimates obtained by plugging the "experimental" values of the proper slices' volumes (see above) into the formula of Corollary 8.1.7.

The programs to conduct the experiments where written in C (using the GNU `gcc-2.8.1` compiler), and run under Solaris 5.7 in a Sun Ultra 5 workstation. AWK scripts where used as a front-end to the C program for easier interaction and for data analysis. The plots were produced with Maple 6 and `gnuplot`.

Part V

CONCLUSIONS AND FUTURE WORK

Throughout this work we have presented two different lines of contributions. The first one is the design of point multidimensional data structures (Part III) while the second one is related to the analysis of associative queries in point multidimensional data structures (Part IV).

Concerning the design of data structures we have introduced two randomized point multidimensional data structures: randomized relaxed K-d trees and randomized quad trees (Chapters 3 and 4) that have several advantages. Their randomized update algorithms are simple to describe and implement, and for any sequence of interleaved insertions and deletions they always produce random relaxed K-d trees and random quad trees, respectively. This implies that the expected cost of associative queries holds independently of the order in which updates are performed. In the case of quad trees they have the additional advantage of being defined for any dimension K.

We have thus shown that the use of randomization provides great flexibility in the design of point multidimensional data structures and that it guarantees the expected costs of associative queries, with simple search and update algorithms.

However, the average case analysis of the split and join operations, which are in the basis of the given randomized update algorithms, is rather complex (and not given in this work) since it implies the solution of a non-linear system of recurrences derived from the interleaved relation existing between them. Since these theoretical results seem difficult to obtain, an experimental analysis could provide useful insights. It could also be interesting to test the performance of both randomized relaxed K-d trees and randomized quad trees against existing specific *benchmarks* as well as against real data.

It is worth noting that K-d trees use only one of the possible attributes of the records as discriminant at each node while quad trees use all the possible ones. In some sense these two data structures are extreme cases of a possible family of hierarchical multidimensional data structures in which the number of attributes used as discriminants in each node is variable, going from one (K-d trees) to K (quad trees). We have some evidence in the sense that formalizing this family of trees and defining randomized update operations on them (which are very similar to the ones for quad trees) can provide a general framework for randomized multidimensional data structures. This general framework would include also randomized structures such as multidimensional skip lists.

We have also introduced in this thesis a new fingering scheme to exploit locality of reference in associative queries on multidimensional data structures.

In particular, we have applied these schemes to K-d trees. For the so called fingered multidimensional trees (Chapter 5) we show that the application of this technique results in a significative amelioration of the performance of hierarchical multidimensional data structures.

A natural extension of this chapter would be to perform the theoretical analysis of the overwork of the cost of range queries and nearest neighbor queries. However, it seems a mathematically challenging problem, mostly because of backtrack. We think that a first step in this direction are the lemmas (Lemmas 5.1.2, 5.1.3, and 5.1.4) that characterize the given search algorithms.

The presented algorithms are straightforwardly applicable to data structures other than K-d trees. For instance, it could be interesting to study the improvements that this technique might yield for nearest neighbor queries in metric data structures [15].

Relative to the analysis of associative queries we present the average-case analysis of partial match queries in random relaxed K-d trees (Chapter 7) as well as the average-case analysis of orthogonal range queries in several hierarchical multidimensional data structures such as: K-d trees, K-d tries, quad trees and quad tries (Chapter 8).

The theoretical analysis, developed in this thesis, shows that the algorithms for associative queries in randomized relaxed K-d trees have reasonable performance on practical grounds. One of the important results of this thesis is the connection between orthogonal range queries and partial match queries. Even though partial match queries are not as useful in applications as range queries and nearest neighbor queries, they are easier to analyze and their analysis does not involve geometric considerations. Here we have shown that, for a large family of hierarchical multidimensional data structures, having the analysis of the cost of partial match queries yields at no cost the average performance of orthogonal range and nearest neighbor queries.

As extension of this line of work, we propose the average-case analysis of range queries but under a different query model. The results of orthogonal range queries presented here depend on a range query model in which the range query have given fixed-length sides and the center of the hyper-rectangle is randomly assigned. The obtained results hold for small hyper-rectangles. We suggest another query model in which the hyper-rectangles are obtained by generating uniformly and independently their corners in $[0,1]^K$. The obtained range queries will be *large* with length sizes of $1/3$ in

average. In the case of random relaxed K-d trees the problem is formally stated in Appendix B.

APPENDIX

A. PLOTS OF FINGERED MULTIDIMENSIONAL TREES

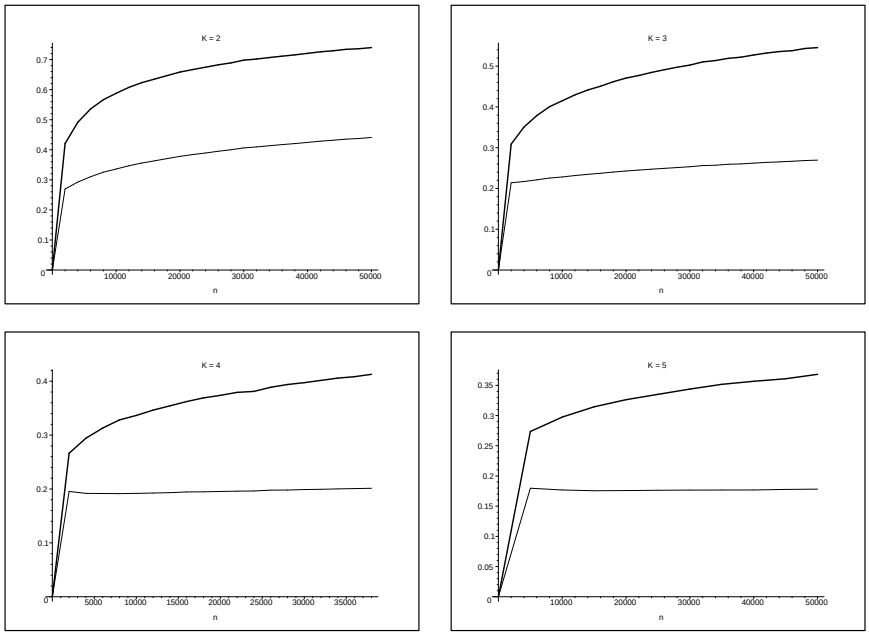


Fig. A.1: Overwork ratios for standard 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.25$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

The graphs of Figures A.1, A.2 and A.3 depict $\tau_n^{(1)}$ and $\tau_n^{(m)}$ for standard 1-finger and m-finger K-d trees for $\delta = 0.25, 0.75$ and 2, respectively. Figure A.4 shows the plots of the overwork of standard m-finger K-d trees for several values of the locality parameter δ.

The behavior of relaxed 1-finger and m-finger K-d trees for orthogonal range searches when the queries have variable length sizes is shown in Figures A.5, A.6 and A.7 for $\delta = 0.25$, 0.75 and, 2 respectively. The graphs corresponding to standard 1-finger and m-finger K-d trees are plotted in Figures A.8, A.9 and A.10. It is worth to note that for range queries with variable length the savings are not as important as for fixed lengths range queries. In fact for δ's greater than 1 there are no savings at all. It is also worth noting that the overwork cost is of the same order of growth as the cost of range queries in plain K-d trees.

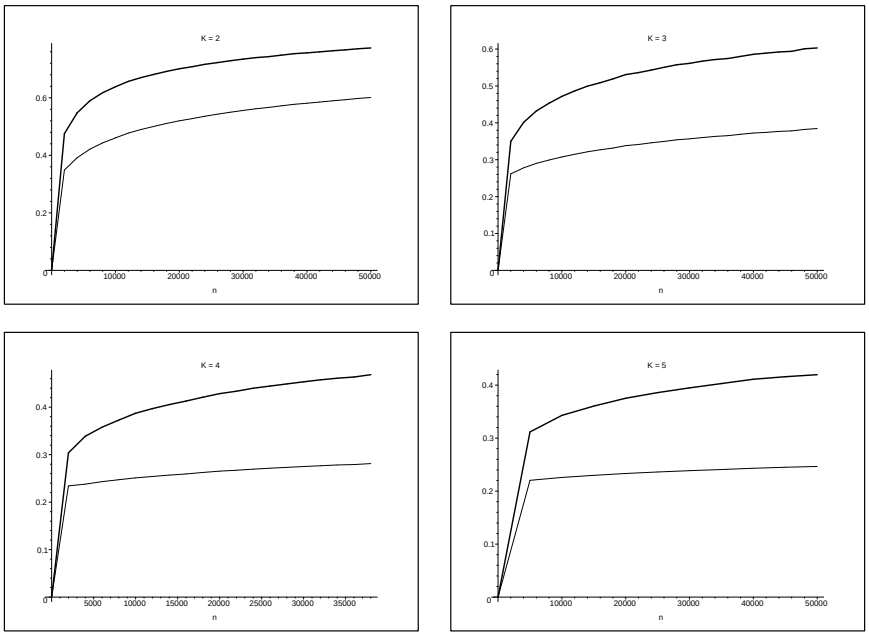


Fig. A.2: Overwork ratios for standard 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.75$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

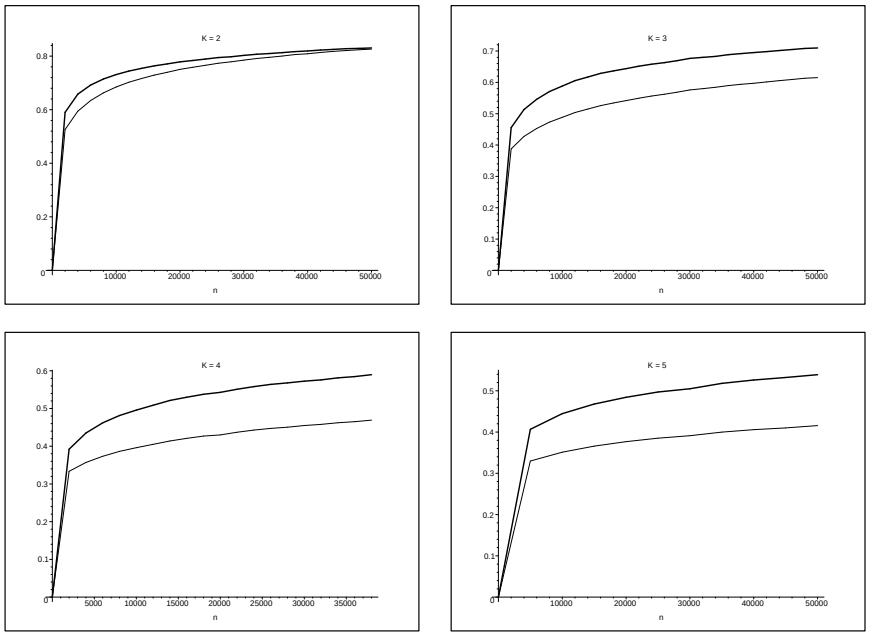


Fig. A.3: Overwork ratios for standard 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 2$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

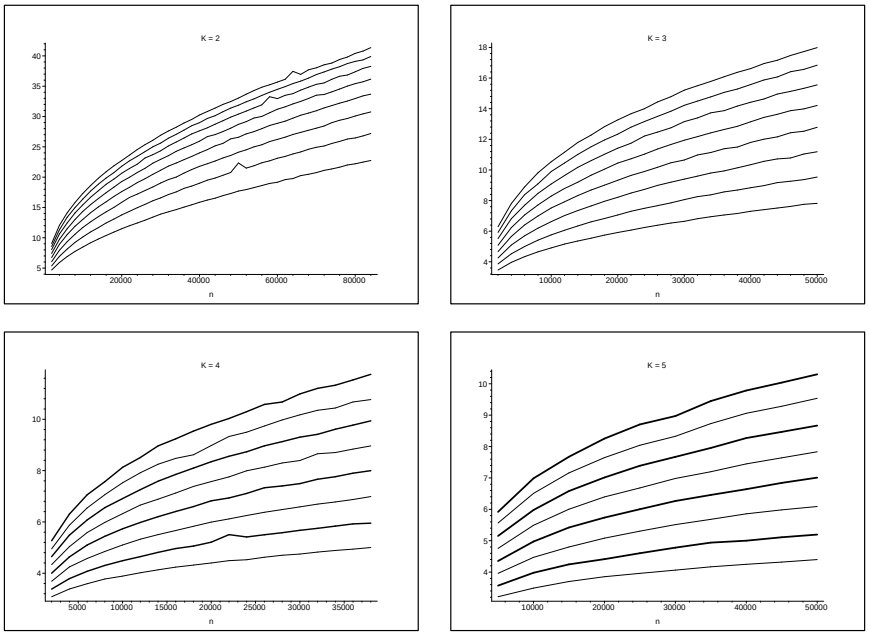


Fig. A.4: Overwork in standard m-finger K-d trees for several values of the locality parameter δ, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

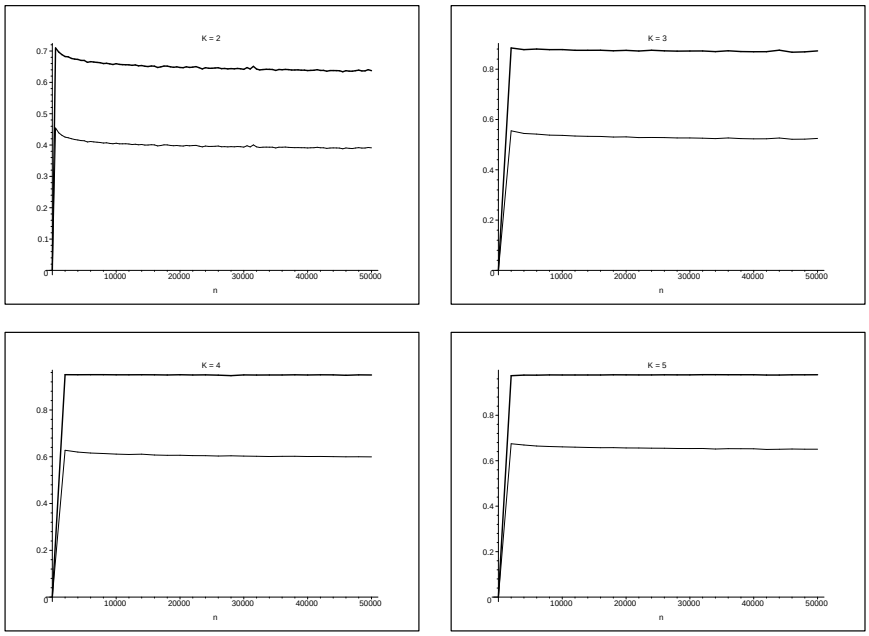


Fig. A.5: Overwork ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.25$, and $\Delta = \sqrt[K]{1/n}$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

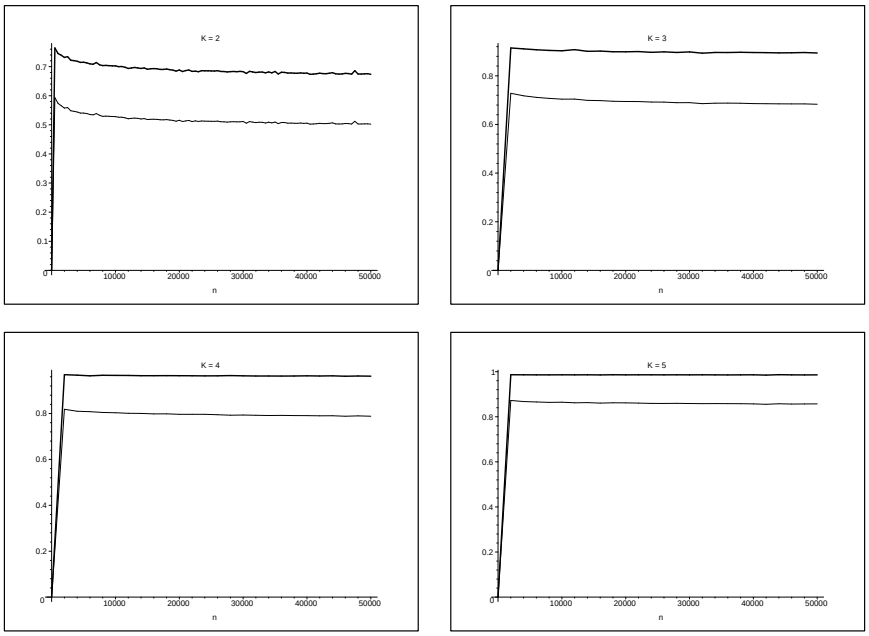


Fig. A.6: Overwork ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.75$, and $\Delta = \sqrt[K]{1/n}$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

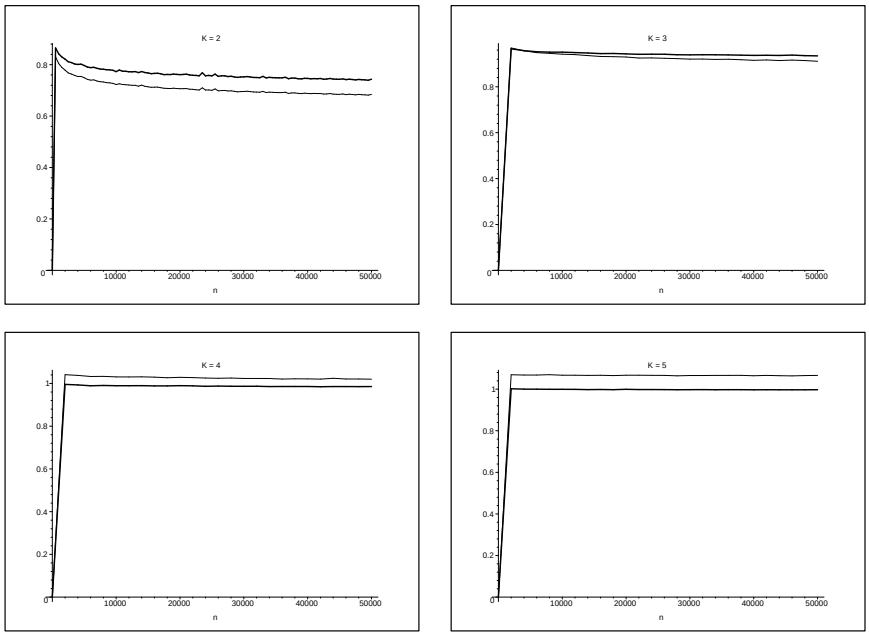


Fig. A.7: Overwork ratios for relaxed 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 2$, and $\Delta = \sqrt[K]{1/n}$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

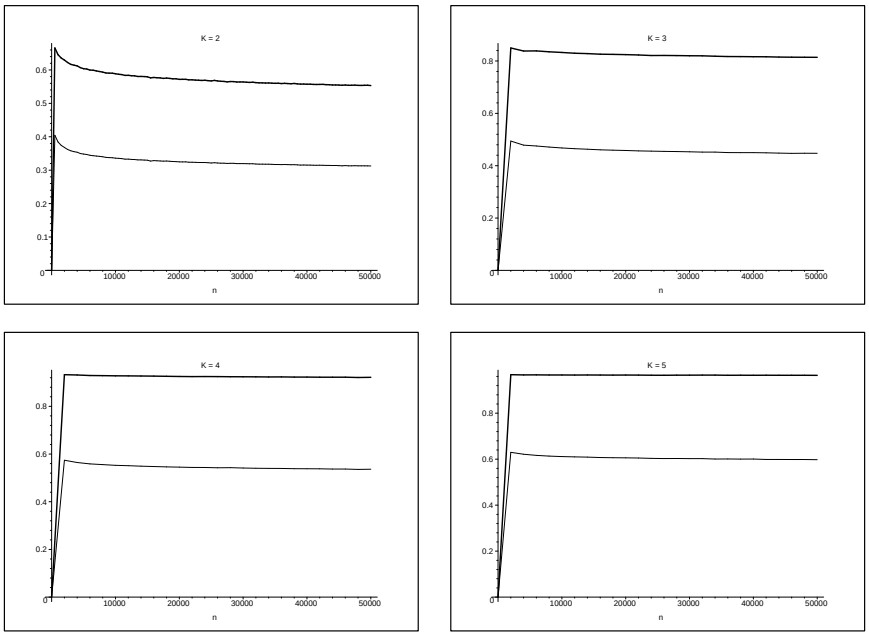


Fig. A.8: Overwork ratios for standard 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.25$, and $\Delta = \sqrt[K]{1/n}$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

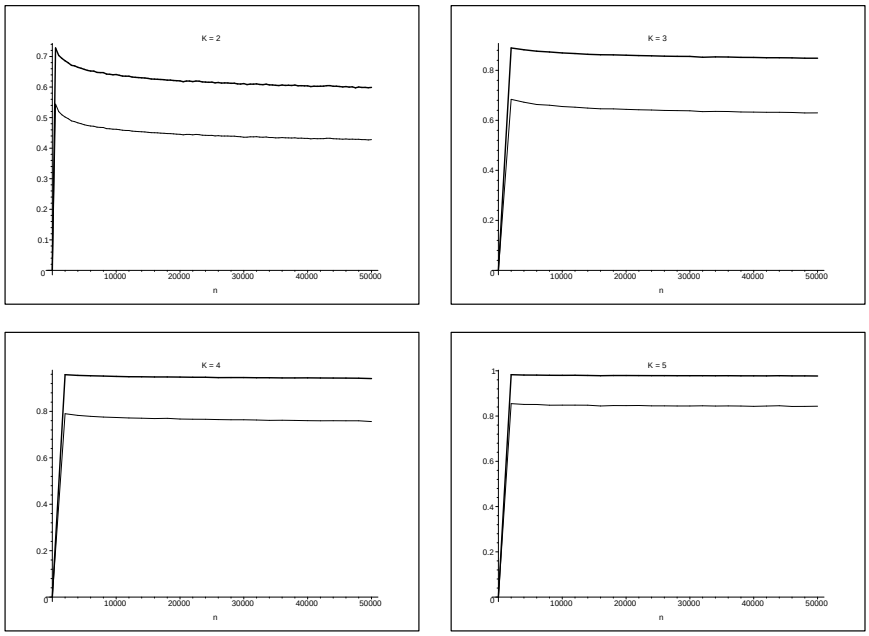


Fig. A.9: Overwork ratios for standard 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 0.75$, and $\Delta = \sqrt[K]{1/n}$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

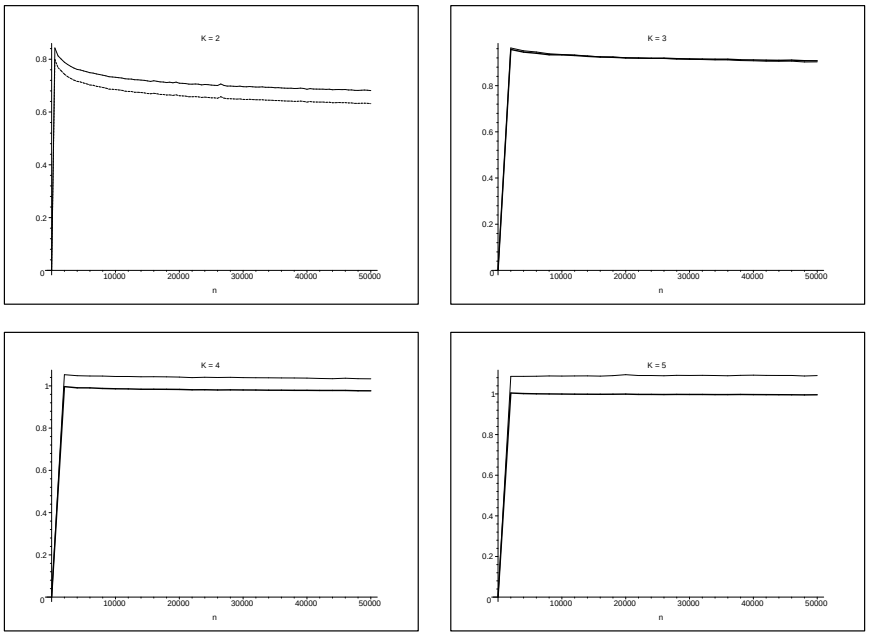


Fig. A.10: Overwork ratios for standard 1-finger K-d trees (*thick line*) and m-finger K-d trees (*thin line*), for $\delta = 2$, and $\Delta = \sqrt[K]{1/n}$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

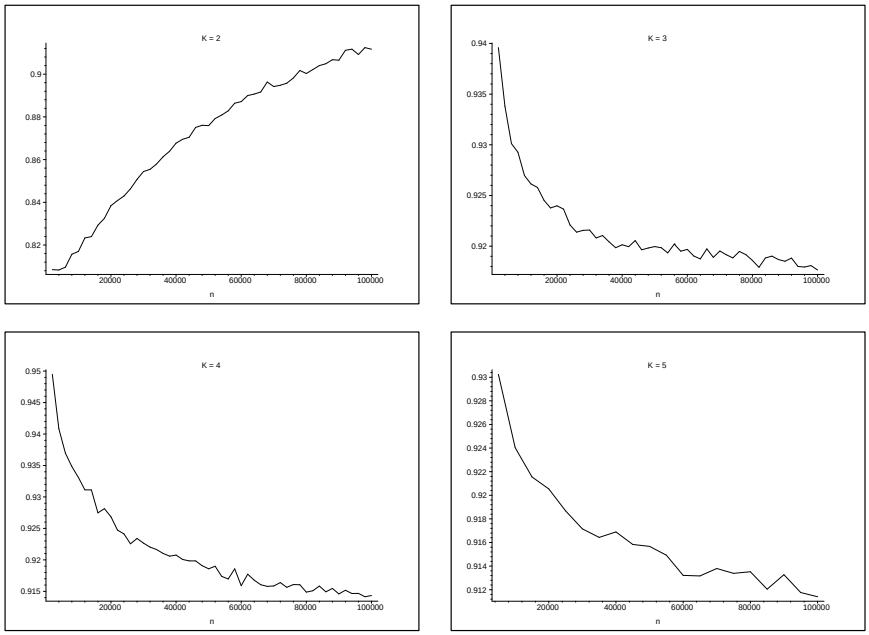


Fig. A.11: Nearest neighbor queries in standard 1-finger K-d trees for $\delta = 0.005$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

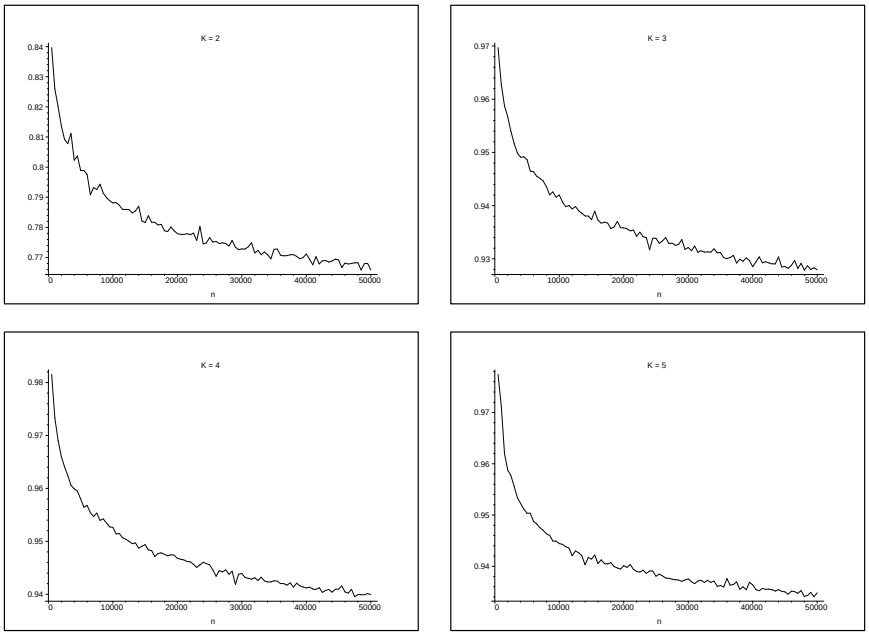


Fig. A.12: Nearest neighbor queries in standard 1-finger K-d trees for $\delta = 0.25\sqrt[K]{1/n}$, $K = 2$ (*up left*), $K = 3$ (*up right*), $K = 4$ (*down left*), and $K = 5$ (*down right*).

B. ANALYSIS OF ORTHOGONAL RANGE QUERIES WITH RANDOM CORNERS

Let $F = \{x^{(0)}, \ldots, x^{(n)}\}$, $n \geq 0$, be the given file (set of K-dimensional points). Each $x \in F$ is thus a K-tuple $x = (x_1, x_2, \ldots, x_{K-1})$, where each x_i belongs to some totally ordered domain D_i. Hence, $x \in D = D_0 \times \cdots \times D_{K-1}$. Let $-\infty$ and $+\infty$ denote special values such that for any $v \in D_i$, $-\infty < v$ and $v < +\infty$, according to the total order defined in D_i, and let $\overline{D}_i = D_i \cup \{-\infty, +\infty\}$, and $\overline{D} = \overline{D}_0 \times \cdots \times \overline{D}_{K-1}$.

Then a range query is a hyper-rectangle $Q \in \overline{D} \times \overline{D} \sim (\overline{D}_0 \times \overline{D}_0) \times (\overline{D}_1 \times \overline{D}_1) \times \cdots (\overline{D}_{K-1} \times \overline{D}_{K-1})$.

Let $Q = (\ell_0, u_0) \times (\ell_1, u_1) \times \cdots \times (\ell_{K-1}, u_{K-1})$, with $-\infty \leq l_i \leq u_i \leq +\infty$, for $0 \leq i < K$. If $l_i = -\infty$ and $u_i = +\infty$ we say that the i-th coordinate of H is unspecified. If $l_i = -\infty$ or $u_i = +\infty$, but not both, we say that the i-th coordinate is half-specified. If $-\infty < l_i \leq u_i < +\infty$, the coordinate is fully-specified.

Let $R_{u,h}(n)$ denote the expected cost (measured as number of visited nodes) in a random range query with u unspecified coordinates and h half-specified coordinates, $0 \leq u \leq K$, $0 \leq h \leq K$, $u + h \leq K$, in a random relaxed K-d tree of size n. Trivially, $R_{u,h}(n) = n$ if $n \leq 1$, for all u and h. Furthermore, for any $n \geq 0$,

$$R_{K,0}(n) = n,$$

since range search reduces to a pre-order traversal of the whole tree when all dimensions are unspecified.

Now, assume $u < K$ and $n > 0$. The random relaxed K-d tree T_n has a left subtree L and a right subtree of sizes k and $n - 1 - k$ with probability $1/n$, for $0 \leq k < n$. If the discriminant of the root node is one of the unspecified coordinates, which happens with probability u/K, then the search continues recursively in both subtrees. On the other hand, if the discriminant, say j, is a half-specified coordinate (with probability h/K) then we have two possibilities: either the value x_j of the element stored in the root is inside the half-specified range or it is not. By symmetry, we might only consider half-specified ranges such that $u_j = +\infty$. The first case occurs then with probability $(k + 1)/(n + 1)$ and we have to continue with the search in both subtrees; however, the range search in the right subtree will be now unspecified with respect to the j-th coordinate. Therefore, we will have a term of the type

$$\frac{k+1}{n+1} \left(R_{u,h}(k) + R_{u+1,h-1}(n-1-k) \right).$$

The second case (x_j fails off the range) occurs with probability $(n-k)/(n+1)$; in this case, we only make one recursive call in the right subtree.

Last but not least, the discriminant can be fully-specified (with probability $1-(u+h)/K$) and we can have three cases: the range is to the left of the root's element along the j-th coordinate ($u_j < x_j$), the range is to the right ($x_j \leq \ell_j$) or the range covers the element ($\ell_j < x_j \leq u_j$). These events happen with probabilities $(k+1)^2/(n+1)^2$, $(n-k)^2/(n+1)^2$ and $2(k+1)(n-k)/(n+1)^2$, respectively. In the first case, we explore only the left subtree, in the second case we explore only the right subtree, and finally, in the third case we explore both subtrees, but now the j-th coordinate is half-specified in the two recursive calls.

Collecting everything the recurrence when $n > 0$ is as follows.

$$\begin{aligned}
R_{u,h}(n) = 1 + \tfrac{1}{n}\textstyle\sum_{0\leq k<n} \Bigg[& \tfrac{u}{K}\left(R_{u,h}(k) + R_{u,h}(n-1-k)\right) \\
& + \tfrac{h}{K}\left(\tfrac{n-k}{n+1}R_{u,h}(n-1-k) + \tfrac{k+1}{n+1}\left(R_{u,h}(k) + R_{u+1,h-1}(n-1-k)\right)\right) \\
& + \left(1-\tfrac{u+h}{K}\right)\Bigg(\left(\tfrac{k+1}{n+1}\right)^2 R_{u,h}(k) + \left(\tfrac{n-k}{n+1}\right)^2 R_{u,h}(n-1-k) \\
& + 2\tfrac{(k+1)(n-k)}{(n+1)^2}\left(R_{u,h+1}(k) + R_{u,h+1}(n-1-k)\right)\Bigg)\Bigg].
\end{aligned}$$

By obvious symmetries the recurrence above can be simplified to,

$$\begin{aligned}
R_{u,h}(n) = 1 & + \tfrac{2u}{nK}\textstyle\sum_{0\leq k<n} R_{u,h}(k) \\
& + \tfrac{2h}{nK}\textstyle\sum_{0\leq k<n} \tfrac{k+1}{n+1}R_{u,h}(k) \\
& + \tfrac{h}{nK}\textstyle\sum_{0\leq k<n} \tfrac{n-k}{n+1}R_{u+1,h-1}(k) \\
& + \tfrac{2(K-(u+h))}{nK}\textstyle\sum_{0\leq k<n} \left(\tfrac{k+1}{n+1}\right)^2 R_{u,h}(k) \\
& + \tfrac{4(K-(u+h))}{nK}\textstyle\sum_{0\leq k<n} \tfrac{(k+1)(n-k)}{(n+1)^2}R_{u,h+1}(k).
\end{aligned}$$

Notice that the recurrence is valid even if $u = K$, since only the first two terms are then non-zero.

For example, take $K = 2$. We are interested in $R_{0,0}(n)$. Using the recurrence above we readily see that we need the value of $R_{0,1}(n)$. To solve the recurrence for $R_{0,1}(n)$ we will have to solve before the recurrences for $R_{1,0}(n)$ and $R_{0,2}(n)$ and so forth. In general, $R_{u,h}(n)$ depends on $R_{u+1,h-1}(k)$, $R_{u,h+1}(k)$ and $R_{u,h}(k-1)$ with $0 \leq k < n$. The graphs of interrelations among the $R_{u,h}$'s is acyclic; about $K^2/2$ recurrences need to be solved to obtain $R_{0,0}(n)$ (which is actually the sought answer).

We have the solutions for the recurrences of the form $R_{K-i,i}$ and $R_{0,i}$ for $i = 0, \ldots, K$ but we have not been able to go any further.

BIBLIOGRAPHY

[1] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proceedings of the 30th IEE Symposium on Foundations of Computer Science(FOCS)*, pages 540–545, 1989.

[2] R. Bayer. The universal B-tree for multidimensional indexing. Technical Report TUM–I9637, Technische Universität München, Munich, Germany, 1996. Available from `http://wwwbib.informatik.tu-muenchen.de/infberichte/1996/`.

[3] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[4] J. L. Bentley. Multidimensional binary search trees used for associative retrieval. *Communications of the ACM*, 18(9):509–517, 1975.

[5] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):133–136, 1979.

[6] J. L. Bentley and R. A. Finkel. Quad trees: A data structure for retrieval on composites keys. *Acta Informatica*, 4:1–9, 1974.

[7] J. L. Bentley, D. F. Stanat, and E. H. Williams. The complexity of finding fixed-radius near neighbors. *Information Processing Letters*, 6(6):209–212, 1977.

[8] J.L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[9] S. Berchtold, D. Keim, and H. P. Kriegel. The X-tree: an index structure for multidimensional data. In *Proceedings of the 22nd. Interna-*

tional Conference on Very Large Databases, (Bombay), pages 28–39, 1996.

[10] H. Blanken, A. Ijbema, P. Meek, and B. van den Akker. The generalized grid file: Description and performance aspects. In *Proceedings of the Sixth IEEE Internatinal Conference on Data Engineering*, pages 380–388, 1990.

[11] A. Borodin and R. El-Yaniv. *On line Computation and Competitive Analysis*. Cambridge University Press, 1998.

[12] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[13] W. A. Burkhard. Hashing and trie algorithms for partial match retrieval. *ACM Transactions on Database Systems*, 1(2):175–187, 1976.

[14] P. Chanzy, L. Devroye, and C. Zamora-Cura. Analysis of range search for random k-d trees. *Acta Informatica*, 37:355–383, 2001.

[15] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[16] H. H. Chern and H. K. Hwang. An asymptotic theory for Cauchy-Euler differential equations with applications to the analysis of algorithms. *Journal of Algorithms*, 44(1):177–225, 2002.

[17] H. H. Chern and H. K. Hwang. Partial match queries in random k-d trees, 2003. Submitted for publication.

[18] H. H. Chern and H. K. Hwang. Partial match queries in random quad trees. *SIAM Journal on Computing*, 32(4):904–915, 2003.

[19] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[20] The Unicode Consortium. *The Unicode Standard Version 2.0, 2d. Edition*. Addison–Wesley, 1996.

[21] J. Culberson. The effect of updates in binary search trees. In *ACM Symp. on the Theory of Computing (STOC'85)*, pages 205–212, 1985.

[22] W. Cunto, G. Lau, and Ph. Flajolet. Analysis of KDT-trees: KD-trees improved by local reorganisations. In F. Dehne, J. R. Sack, and N. Santoro, editors, *Workshop on Algorithms and Data Structures (WADS'89)*, volume 382 of *LNCS*, pages 24–38. Springer–Verlag, 1989.

[23] L. Devroye. Branching processes in the analysis of the height of trees. *Acta Informatica*, 24:277–298, 1987.

[24] L. Devroye, J. Jabbour, and C. Zamora-Cura. Squarish k-d trees. *SIAM Journal on Computing*, 30:1678–1700, 2000.

[25] L. Devroye and L. Laforest. An analysis of random d-dimensional quadtrees. *SIAM Journal on Computing*, 19(5):821–832, 1990.

[26] A. Duch. Randomized insertion and deletion in point quad trees. In *Int. Symposium on Algorithms and Computation (ISAAC)*, LNCS. Springer–Verlag, 2004. Accepted for publication.

[27] A. Duch, V. Estivill-Castro, and C. Martínez. Randomized K-dimensional binary search trees. In K.-Y. Chwa and O. H. Ibarra, editors, *Int. Symposium on Algorithms and Computation (ISAAC'98)*, volume 1533 of *LNCS*, pages 199–208. Springer-Verlag, 1998.

[28] A. Duch, V. Estivill-Castro, and C. Martínez. Randomized K-dimensional binary search trees. Technical Report LSI–98–48–R, LSI-UPC, 1998.

[29] A. Duch and C. Martínez. On the average performance of orthogonal range search in multidimensional data structures. In P. Widmayer, F. Triguero, R. Morales, R. Henessy, S. Eidenbenz, and R. Conejo, editors, *Proc. of the 29th. Int. Col. on Automata, Languages and Programming (ICALP)*, volume 2380 of *LNCS*, pages 514–524. Springer–Verlag, 2002.

[30] A. Duch and C. Martínez. On the average performance of orthogonal range search in multidimensional data structures. *Journal of Algorithms*, 44(1):226–245, 2002.

[31] A. Duch and C. Martínez. Fingered multidimensional search trees. In C. C. Ribeiro and S. L. Martins, editors, *Proc. of the Third International Workshop on Experimental and Efficient Algorithms (WEA)*, volume 3059 of *LNCS*, pages 228–242. Springer–Verlag, 2004.

[32] J. L. Eppinger. An empirical study of insertion and deletion in binary search trees. *Communications of the ACM*, 26(9):663–669, 1983.

[33] G. Evangelidis, D. Lomet, and B. Salzberg. The hB^{π}-tree: a modified hB-tree supporting concurrency, recovery and node consolidation. In *Proceedings of the 21st International Conference on Very Large Databases*, pages 551–561, 1996.

[34] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite key. *Acta Informatica*, 4(1):1–9, 1974.

[35] Ph. Flajolet, G. Gonnet, C. Puech, and J. M. Robson. Analytic variations on quad trees. *Algorithmica*, 10:473–500, 1993.

[36] Ph. Flajolet and T. Lafforgue. Search costs in quad trees and singularity perturbation analysis. *Discrete and Computational Geometry*, 12(4):151–175, 1993.

[37] Ph. Flajolet and A. Odlyzco. Singularity analysis of generating functions. *SIAM Journal of Discrete Mathematics*, 3(1):216–240, 1990.

[38] Ph. Flajolet and C. Puech. Partial match retrieval of multidimensional data. *Journal of the ACM*, 33(2):371–407, 1986.

[39] M. Freeston. A general solution of the n-dimensional B-tree problem. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 80–91, 1995.

[40] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[41] M. Friedman, F. Baskett, and L. J. Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on Computing*, C–24(10):1000–1006, 1975.

[42] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3), 1980.

[43] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[44] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics, 2nd edition.* Addison–Wesley, 1994.

[45] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Symp. on the Theory of Computing (STOC)*, 1977.

[46] O. Günter. *Efficient Structures for Efficient Data Management*, volume 337 of *LNCS*. Springer–Verlag, 1988.

[47] O. Günter. Evaluation of spatial access methods with oversize shelves. In G. Gambosi, M. Scholl, and H. Six, editors, *Geographic Database Management Systems*, pages 177–193. Springer–Verlag, 1991.

[48] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. of the ACM–SIGMOD Annual Meeting*, 14(2):47–54, 1984.

[49] A. Henrich and J. Moller. Extending spatial access structures to support additional standard attributes. In M. J. Egenhofer and J. R. Herring, editors, *Proceedings of the 4th Int. Symp. on Advances in Spatial Data Bases*, volume 951 of *LNCS*, pages 132–151. Springer–Verlag, 1995.

[50] A. Henrich, H. W. Six, and P. Widmayer. The *LSD* tree: Spatial access to multidimensional point and non-point objects. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 45–53, 1989.

[51] P. Henrici. *Applied and Computational Complex Analysis, volume 1.* Wiley-Interscience, 1974.

[52] P. Henrici. *Applied and Computational Complex Analysis, volume 2.* Wiley-Interscience, 1977.

[53] K. Hinrichs. Implementation of the grid file: design, concepts and experience. *BIT*, 25:569–592, 1985.

[54] A. Hutflesz, H. W. Six, and P. Widmayer. Twin grid files: Space optimizing access schemes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 183–190, 1988.

[55] H. V. Jagadish. Spatial search with polyhedra. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 311–319, 1990.

[56] I. Kamel and C. Faloutsos. Parallel *R*-trees. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 195–204, 1992.

[57] I. Kamel and C. Faloutsos. Hilbert *R*-trees: An improved *R*-tree using fractals. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 500–509, 1994.

[58] J. S. Karlsson and M. L. Kersten. Omega-storage: A self organizing multi-attibute storage technique for very large main memories. *Report of the Centrum voor Wiskunde en Informatica*, 1999.

[59] P. Kirschenhofer and H. Prodinger. Multidimensional digital searching in alternative data structures. *Random Structures and Algorithms*, 5(1):123–134, 1994.

[60] D. E. Knuth. *The art of computer programming, vol. 3: sorting and searching, 2nd. edition.* Addison–Wesley, Reading, Mass., 1999.

[61] H. P. Krieger and B. Seeger. Multidimensional order preserving linear hashing with partial expansions. In *Proceedings of the International Conference on Database Theory*, volume 243 of *LNCS*. Springer–Verlag, 1986.

[62] H. P. Krieger and B. Seeger. Multidimensional quantile hashing is very efficient for non-uniform record distributions. In *Proceedings of the Third IEEE International Conference on Data Engineering*, pages 10–17, 1987.

[63] H. P. Krieger and B. Seeger. PLOP-hashing: a grid file without directory. In *Proceedings of the Fourth IEEE International Conference on Data Engineering*, pages 369–376, 1988.

[64] H. P. Krieger and B. Seeger. Multi dimensional quantile hashing is very efficient for non-uniform distributions. *Information Science*, 48:99–117, 1989.

[65] A. Kumar. G-tree: A new data structure for organizing multidimensional data. *IEEE Trans. Knowl. Data Eng.*, 6(2):341–347, 1994.

[66] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad-trees. *Acta Informatica*, 9(1):23–29, 1977.

[67] R. C. T. Lee, T. H. Chin, and S. C. Chang. Application to principal component analysis to multi–key searching. *IEEE Transactions on Software Eng.*, SE-2(3):185–193, 1976.

[68] K. I. Lin, H. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB J.*, 3(4):517–543, 1994.

[69] D. B. Lomet and B. Salzberg. The hB-tree: A robust multi attribute search structure. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 296–304, 1989.

[70] H. M. Mahmoud. *Evolution of Random Search Trees.* Wiley–Interscience Series. John Wiley and Sons, 1992.

[71] C. Martínez, A. Panholzer, and H. Prodinger. On the number of descendants and ascendants in random search trees. *Electronic Journal on Combinatorics*, 5(1):181–204, 1998.

[72] C. Martínez, A. Panholzer, and H. Prodinger. Partial match queries in relaxed multidimensional search trees. *Algorithmica*, 29(1–2):181–204, 2001.

[73] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[74] T. Matsuyama, L. Hao, and M. Nagao. A file organization for geographic information systems based on spatial proximity. *J. Comput. Vis. Graph. Image Process.*, 26(3):303–318, 1984.

[75] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[76] K. Mulmuley. Randomized multidimensional search trees: further results in dynamic sampling. *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 216–227, 1991.

[77] K. Mulmuley. Randomized multidimensional search trees: lazy balancing and dynamic shuffling. *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 180–196, 1991.

[78] R. Neininger. *Limit Laws for Random Recursive Structures and Algorithms*. Dissertation, Universität Freiburg i. Br., 1999.

[79] R. Neininger. Asymptotic distributions for partial match queries in K-d trees. *Random Structures and Algorithms*, 17(3–4):403–4027, 2000.

[80] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable symmetric multikey file structure. *ACM Transactions on Database Systems*, 1(9):38–71, 1984.

[81] Y. Ohsawa and M. Sakauchi. *BD*-tree: A new n-dimensional data structure with efficient dynamic characteristics. In *Proceedings of the Ninth World Computer Congress, IFIP*, pages 539–544, 1983.

[82] Y. Ohsawa and M. Sakauchi. A new tree type data structure with homogenous node suitable for a very large spatial database. In *Proceedings of the Sixth IEEE International Conference on Data Engineering*, pages 296–303, 1990.

[83] B. C. Ooi, K. J. McDonell, and R. Sacks-Davis. Spatial *kd*-tree: An indexing mechanism for spatial data bases. In *Proceedings of the IEEE Computer Software and Applications Conference*, pages 433–438, 1987.

[84] P. Oosterom. *Reactive Data Structures for Geographic Information Systems*. PhD Thesis, University of Leiden, The Netherlands., 1990.

[85] J. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Data base Systems*, pages 181–190, 1984.

[86] M. H. Overmars and J. van Leeuwen. Dynamic multi-dimensional data structures based on quad- and k-d-trees. *Acta Informatica*, 17(3):267–285, 1982.

[87] W. Pugh. Skip lists: A probabilistic alternative to balance trees. *Communications of the ACM*, 33(6):668–676, 1990.

[88] M. Regnier. Analysis of the grid file algorithms. *BIT*, 25(2):335–357, 1985.

[89] R. L. Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[90] J. T. Robinson. The KDB-tree: A search structure for large multidimensional dynamic indexes. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[91] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 17–31, 1985.

[92] H. Sagan. *Space Filling Curves.* Springer–Verlag, 1994.

[93] H. Samet. Deletion in two-dimensional quad-trees. *Communications of the ACM*, 23(12):703–710, 1980.

[94] H. Samet. *Applications of Spatial Data Structures.* Addison–Wesley, 1990.

[95] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison–Wesley, 1990.

[96] R. Schneider and H. P. Kriegel. The TR^*-tree: A new representation of polygonal objects supporting spatial queries and operations. In *Proceedings of the Seventh Workshop on Computational Geometry*, volume 553 of *LNCS*, pages 249–264. Springer–Verlag, 1992.

[97] B. Seeger and H. P. Kriegel. Techniques for design and implementation of spatial access methods. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, pages 360–371, 1988.

[98] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the Thirteenth International Conference on Very Large Databases*, pages 507–518, 1987.

[99] Y.V. Silva-Filho. Average case analysis of region search in balanced k-d trees. *Information Processing Letters*, 8(5):219–223, 1979.

[100] D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[101] R. Sproull. Refinements to nearest neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[102] M. Tamminen. The extendible cell method for closest point problems. *BIT*, 22:27–41, 1982.

[103] M. J. van Kreveld and M. H. Overmars. Divided k-d trees. *Algorithmica*, 6(6):840–858, 1991.

[104] J. S. Vitter and Ph. Flajolet. *Average-case analysis of algorithms and data structures. In J. van Leewen, editor, Handbook of Theoretical Computer Science, volume A, chapter 9.* North–Holland, 1990.